University of Oslo
Department of Informatics

Development of a roaming real-time patient monitor

Fredrik de Vibe

Cand. Scient. Thesis

1st November 2005

Abstract

The goal of this thesis was to develop an application facilitating patient monitoring by providing a roaming monitor that a physician can carry with him or her at any time. The thesis itself presents an overview of issues encountered before and during the development of the application and explains in detail how and why it was developed. Central topics are embedded software design and implementation and pattern recognition in medical signals.

Contents

1 Introduction **3**

1.1 Background 3

1.2 Objectives 4

1.3 Introduction to some terms 5

1.4 Chapter overview 5

2 Background research and ideas **7**

2.1 Some challenges in the ICU 7

2.2 Preliminary ideas 8

2.2.1 Scoring systems 8

2.2.2 Journal access 9

2.2.3 Prescription approval 10

2.2.4 Communication 10

2.2.5 Patient monitor 10

3 Hardware and software alternatives **13**

3.1 Network protocol 13

3.1.1 A UDP / RTP scenario 15

3.1.2 A TCP scenario 15

3.1.3 Network protocol conclusion 16

3.2 Networking technology 16

3.2.1 IrDA 17

3.2.2 Bluetooth 17
3.2.3 IEEE 802.11 17
3.2.4 Health implications 17
3.2.5 Networking technology conclusion 18
3.3 Operating system 19
3.3.1 Palm OS 19
3.3.2 Windows 19
3.3.3 Linux 20
3.3.4 Operating system conclusion 20
3.4 Programming language 21
3.4.1 C 21
3.4.2 C++ 22
3.4.3 Java 22
3.4.4 Python 23
3.4.5 LabVIEW 23
3.4.6 Language conclusion 23
3.5 Windowing toolkit 24
3.6 Hardware device 25
3.6.1 The future of PDAs 27
3.7 Hardware and software summary 27

4 Embedded Linux **29**
4.1 Boot sequence of Linux systems 30
4.2 File systems 31
4.2.1 A Flash memory file system 32
4.3 Running Linux on the HP iPAQ H5450 33
4.3.1 Preparations for installation 35
4.3.2 Installing Linux 35

4.3.3 Setting up multiple systems with the monitoring application 36

5 Software compilation **39**

5.1 Native compilation 40

5.2 Cross compilation 40

5.3 Compiling software for embedded devices 41

5.3.1 Compiling natively on embedded devices 41

5.3.2 Cross compiling for embedded devices 42

5.4 Compilation experiences 44

6 Automated pattern recognition in ECG signals **47**

6.1 QRS detection 49

6.1.1 Modern algorithms 50

6.1.2 Real time electrocardiogram QRS detection using combined adaptive threshold 52

6.1.3 Implementation 55

7 Software design **59**

7.1 Processing and displaying data 60

7.2 Client only scenario 63

7.2.1 Computationally intensive model 63

7.2.2 Computationally simple model 64

7.3 Client — server scenario 65

8 Tests and results **67**

8.1 Benchmark tests 69

8.1.1 Benchmarking memory 70

8.1.2 Benchmarking CPU 73

8.1.3 Benchmarking conclusions 77

9 Conclusion and future work **79**

9.1 The monitoring application 79

9.2 Platform and operating system 80

9.3 Further work 81

List of Figures

1.1 Front side of an ICU form from Rikshospitalet. 4

2.1 Remotely monitoring a patient. 11

3.1 A network topology, with node *A* about to communicate with node *B*. 13

3.2 The network from Figure 3.1, using UDP and TCP. . . . 14

5.1 Building a cross compilation toolchain. 41

5.2 iPAQ / Skiff cluster. 43

6.1 ECG model. 48

6.2 The human heart[18]. 49

6.3 12 lead ECG. 50

6.4 Approximate placement of electrodes for standard 12-lead ECG. 51

6.5 Adaptive steep-slope threshold. 53

6.6 Adaptive integrating threshold. 54

6.7 Adaptive beat expectation threshold. 54

6.8 FIR smoothing filter. 55

6.9 Example of using a smoothing filter to reduce interference. 56

6.10 Calculated complex lead. 57

7.1 Screenshot of monitoring application. 61

7.2 A plot from the monitoring application 61
7.3 A closer look at a plot. 62
7.4 Updating the plot 62
7.5 Client receives, processes and presents raw data. . . . 64
7.6 Dedicated server performing most calculations. 66

8.1 Early version of monitoring application in a client-only scenario. 68
8.2 Results from memory testing. (System specifications can be found in Table 8.1.) 71
8.3 Results from CPU testing using a prime number generating algorithm. 75
8.4 Calculating sin(1). 76
8.5 Calculating prime numbers not using the FPU. 76

Preface

Acknowledgements

A great many people have helped me in the work with this thesis, and I wish to thank everyone who has helped me see it through.

At Rikshospitalet, I would like to thank Jon Fredrik Stuestøl, who helped me get in touch with the people who gave me this assignment, Jan Olav Høgetveit, Ola Sveen, Ansgar Oddne Aasen, and the rest of the people at Institute for Surgical Research, who helped me form the idea and helped me further with more contacts as well as followed up on me. Audun Stubhaug and Fridtjov Riddervold as well as the other people at *Generell Intensiv* (Intensive Care Unit, or ICU) provided me with invaluable insight into the daily life in the ICU. Håkon Haugtomt gave me access to state-of-the-art monitoring equipment. Karl Øyri, Ilangko Balasingham, Eigil Samset and the team at The Interventional Centre provided me with equipment, a place to work and professional guidance.

At the Department of Informatics, University of Oslo, my supervisor, Dag Langmyhr has been a great support and always been available with wise counsel. Igor Valerjevich Rafienko has been a good friend and an invaluable programming support without whom my C++ skills and the resulting code would be much poorer. I also want to thank Thorkild Stray and Henrik Grindal Bakken for great help with proofreading, this thesis would not be the same without it.

Even more people have helped me with the thesis. On the *freenode* IRC (Internet Relay Chat) network, there are several channels dedicated to different sides of embedded Linux. I would like to thank the people at #oe (OpenEmbedded), #handhelds.org,

#familiar, #opie, and #gpe, as well as all the other channels I've visited for help with particular problems. Finally, I also need to thank my former colleagues from the computer lab at *Niels Henrik Abels hus*, who, almost always available on IRC, have provided much assistance, 24 hours a day.

I also wish to thank my wife and former fellow student, Ingrid Chieh Yu, who has helped me with proofreading and many ideas, besides a motivation without which this thesis could never have been completed and a patience without which I would have had a difficult time during the last months of work.

Finally, all my family and friends may have noticed my almost complete absence during the final months of work on the thesis; they all deserve warm thanks for their understanding and patience and for still being there.

Notes on the Cambridge Research Laboratory

In 1987, Compaq founded the Cambridge Research Laboratory (CRL)[25]. After a merger with Hewlett Packard (HP) in 2002, CRL has continued to exist and their research into open source software has benefited greatly to this thesis. CRL has provided much of the guidelines and software used for running Linux on iPAQ PDAs and they have run the *handhelds.org* community, hosting a web site with an abundance of information about running open source software on handheld devices.

On October 21st, the CRL was discontinued by HP. At that time, this thesis was one week short of completion, and the future of the handhelds.org community was somewhat uncertain. The community's servers, however, have been moved to the *Open Source Laboratory* at the University of Oregon (http://osuosl.org/), but the exact form of the community in the near future is hard to predict. Some references to the handhelds.org community in this thesis might therefore become outdated, but as the interest for open source development on handheld devices is great, it is probable that information and software will still be available, although maybe not in the exact same places and forms.

Chapter 1

Introduction

1.1 Background

Modern hospitals use vast amounts of technical equipment, such as instruments for measuring heart rate, respiration, brain activity etc. Traditionally, these data have been displayed on separate monitors and have been collected on large sheets of paper by medical nurses and doctors (see Figure 1.1 on the following page). This work is tedious and error prone, and as a result, new systems are emerging that aim to collect this data and display it in the best way possible, using statistics and *scoring systems* (described in 2.2.1 on page 8) to provide diagnostic material.

To this date, few people would say that this way of making diagnoses or monitoring a patient provides a replacement for nurses and physicians. Modern medicine may use more research and statistical data and textbook knowledge than before, but there is still no replacement for the trained eye and mind of a good physician. Computers with automatic diagnostic software may provide good aids, but as they run the risk of stealing the attention of doctors and nurses from the patient, this software should be carefully crafted to be as little intrusive as possible.

The work in a hospital is highly dynamic, with nurses and doctors running from bed to bed and department to department. Thus, the time to review each patient is limited and this should be performed as efficiently as possible for the ward and hospital to be run efficiently. One way to provide quick information about the state of patients would be for a doctor to constantly be within arm's

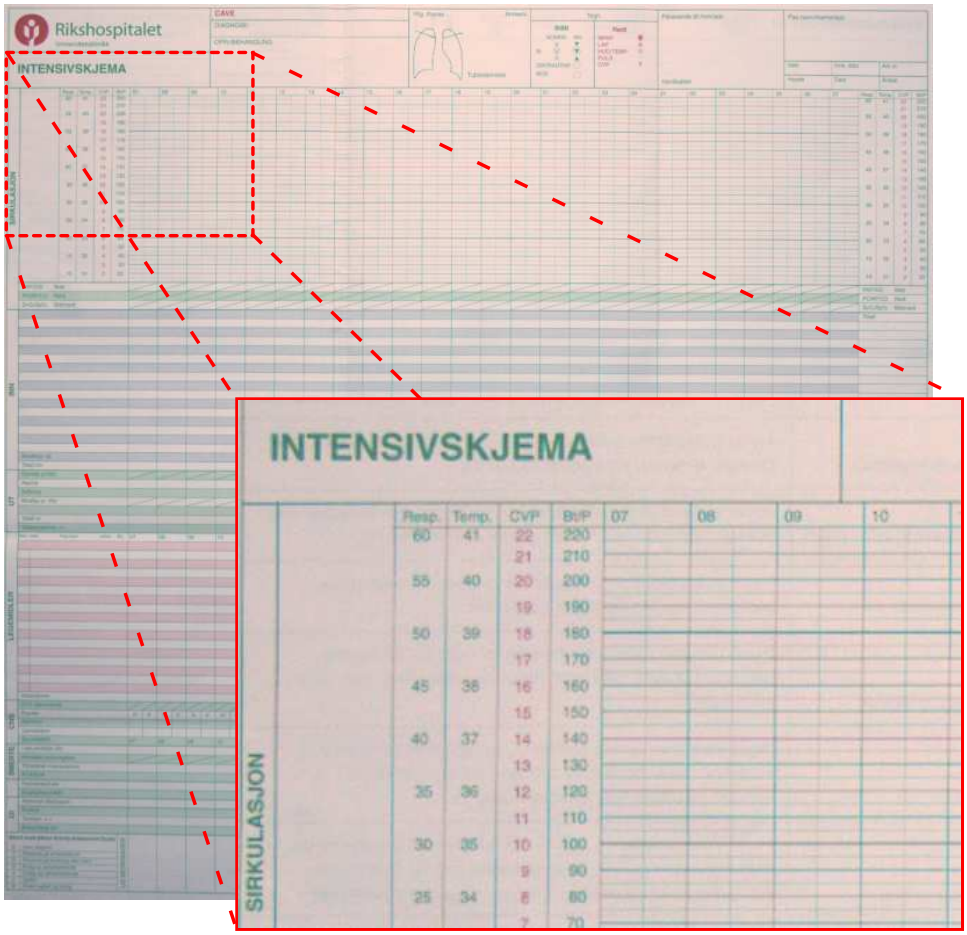


Figure 1.1: Front side of an ICU form from Rikshospitalet. The form is 51.5x50 cm and logs the condition of a patient during 24 hours. Data is plotted in the enlarged area and written into most of the rest of the form.

length of data about the patient's current condition. In a setting with a wireless network covering a hospital, a doctor could use a *Personal Digital Assistant (PDA)* connected to a wireless network to constantly receive data about patients enabling him or her to quickly assess any situation that could arise.

1.2 Objectives

The purpose of this thesis is to develop a proof-of-concept type of PDA application that can be used to monitor patients, both

automatically as in providing possibilities to set alarm levels at which the PDA can alert the user of some condition requiring immediate attention, and manually, so a physician can update him- or herself on a patient's condition without having to cease or pause other activities.

There are immense amounts of combinations of signals to measure, ways in which to measure them and ways in which to combine them, programming tools, hardware and software to use. This thesis focuses on the procedure of developing such an application, the choices that needed to be made and their outcome.

1.3 Introduction to some terms

In this thesis, *workstation* is used as a collective term identifying desktop computers, laptop computers, and servers. An *embedded system* is, on the other hand, defined to be "a special purpose computer system, which is completely encapsulated by the device it controls" [16]. Mobile telephones, modern television sets, anti-lock systems for brakes in modern cars, and PDAs are a few examples of the myriad of embedded systems that exist.

The proof-of-concept type of roaming real-time patient monitoring application that has been developed as the basis for this thesis will normally be referred to as the *monitoring application*.

1.4 Chapter overview

This thesis is divided into chapters in the following manner:

- This chapter gives a short introduction to the thesis and presents an outline of how it is structured.
- In Chapter 2, the basis for the thesis is explained in larger detail, along with initial ideas about options the monitoring application could be equipped with.
- Chapter 3 presents a variety of software and hardware alternatives that were considered for the development. The conclusions are presented along with the options.

- Installing and running Linux on an unsupported platform is not always a trivial task, and in general, running Linux on embedded systems presents some challenges not present on workstations. Chapter 4 gives an introduction to this subject and looks at the experiences made while working on this thesis.
- In Chapter 5, software compilation for embedded devices is studied.
- Automatic interpretation of data is the subject of Chapter 6, in particular, detection of QRS complexes in ECG data, providing the pulse rate.
- The ideas that lie behind how the monitoring application was developed, as well as the development procedure and choices made underway are explained in Chapter 7.
- Chapter 8 presents tests and comparisons of the PDA model used for development and gives some guidelines about what types of operations it is best suited for.
- Finally, in Chapter 9, we summarise the results of this thesis and explore opportunities for further development and work.

Chapter 2

Background research and ideas

In order to acquire a sufficient understanding of the issues relating to patient data representation, I was allowed to visit an *ICU — Intensive Care Unit —* at *Rikshospitalet* in Oslo. For two days, I followed physicians and nurses in their daily work and thus got a glimpse into how their work is performed and what challenges it provides on a daily basis.

2.1 Some challenges in the ICU

ICUs are probably among the busiest departments in hospitals. The patients there are, more often than not, seriously ill, and need to be monitored constantly, 24 hours a day. Most patients have their own dedicated nurse that follow him or her at all times, noting measurements from instruments and every change in condition on large sheets (see Figure 1.1), all the time making adjustments to patients' dosages of medication.

During my short visit to the ICU, I witnessed a hectic activity. The number of patients vary, but even with only about half the beds filled, rigid control is necessary. Many patients in the ICU are seriously ill and their conditions can change quickly. Maintaining accurate records at all times is crucial for correctly assessing the given patients' conditions, and maintaining these records are mostly done by the nurses who are constantly watching over the patients. Physicians roam and check on different patients during a shift and have only limited time for each patient.

The physicians responsible for the patients are mostly from other departments of the hospital and are specialists in different areas. The patients often come to the ICU after surgery to be kept under strict observation. When the condition of a patient deteriorates, the responsible physician is normally consulted and will often attend a quick meeting with the ICU physicians and nurses caring for the patient. This responsible physician might be busy with other patients and is likely to be in entirely different parts of the hospital. When called upon, this physician can for example be in the middle of surgery, causing a prolonged time to wait for assessing the ICU patient's situation.

The conditions of patients can be caused by many factors, and even specialists in a given field can have difficulties determining the correct action at any given time. From my point of view, quick access to accurate information about the state of the patient is crucial as decisions must often be made quickly, and considering the amount of information available, good automatic interpretation of this information could potentially be able to reduce the assessing time. Another factor is the time it takes to get to the information. As the fast pace in a stressed situation makes it necessary for the responsible physician to go straight to the ICU as soon as possible, he or she will not have time or opportunity to review the patient's condition until they have arrived. A possibility to become updated on the patient's condition before this point could also be able to increase the efficiency of assessing a patient's condition.

2.2 Preliminary ideas

The following is a collection of ideas of how a PDA connected via wireless technology to a hospital's computer network could provide useful functionality both in the decision making process and in more administrative fields, potentially easing the workload of hospital personnel.

2.2.1 Scoring systems

Although the primary incentive to move from manual to electronic patient status records was likely a wish to simplify the work

for nurses and possibly to decrease the possibility of human error, one field emerges as a particularly useful one. Using a computer to combine multiple realtime measurements from different instruments may provide valuable information and trends that are hard to spot for a human eye. This is a sort of pattern recognition, where predefined values from a predefined data set can describe a patient's health status at any time.

Scoring systems is an example of how data measurements are combined with demographic data to produce a score calculating the hospital mortality rate for patients. Examples of such systems in use today are *APACHE II*, *MPM II* and *SAPS II*, where the latter is used in Norway. While these systems provide valuable information about a patient's health condition, they are developed to be "static" evaluation systems in the sense that they are calculated only once, based on measurements taken during the patient's first 24 hours in the ICU.

The static nature of scoring systems prevents the scores from being used directly in continuous patient monitoring, but in [8], scoring systems and *early markers* are incorporated in making diagnostic predictions. An early marker for a given condition is an indication that it is likely to occur, or imminent, and serves as a warning. Instead of focusing on the calculated score, this system uses the development in the different factors used in score calculation as an indication for the development of the patient's condition. An application was developed in [8], using the *LabVIEW* programming language, with the intent of being an *early warning* system. This early warning system aims to be able to predict certain conditions, like *sepsis related organ failure* and *multiple system organ failure*. The idea is promising, but research in this area is limited. Depending on future results, an approach like this should be explored further.

2.2.2 Journal access

All information about every patient is logged in the patient's journal. This includes administrative and demographic data, like name, address, social security number etc., but also clinical data, like age, sex, test data, measurement data, diagnoses, notes about a patient's condition etc. Roaming access to these journals could be practical for physicians, as they would not need to go to an office

or a stationary computer to retrieve pieces of information about specific patients or make notes or corrections.

The ICU forms constantly updated by the ICU staff (see Figure 1.1) are used while assessing patients' condition and contain much valuable information. Some, or all, of this information transferred to a portable device could be of use for a physician moving from some other department to the ICU, and could possibly even make such a trip superfluous.

2.2.3 Prescription approval

All prescriptions and many decisions, like changing respirator settings, must be authorised by clinical personnel permitted to make such decisions. Normally, this is a bureaucratic procedure, where a physician has to sign an order explicitly. This could be done using a PDA which a doctor could keep with him or her, where requests for approval could be heralded by for instance a sound signal and approval (or rejection) could be given instantaneously.

2.2.4 Communication

In order to get in touch with physicians, many hospitals use pagers, or beepers. *Short message service* (SMS) on mobile telephones has become a very popular communication channel, and a portable device connected to a wireless network could provide a new way of getting messages to physicians in transit.

One possibility is that results from laboratory tests could be fetched from the hospital network over a portable device, and requests for new tests could be made without having to fill in forms or go to a stationary computer.

2.2.5 Patient monitor

The most important area, and the focus of this thesis, is that a PDA can be used as a roaming real-time patient monitor. In ICUs, patients are constantly monitored by a variety of equipment with dedicated monitors, and normally, one nurse follows a patient at

all times[1]. This means that changes in condition will be noticed quickly and can be responded to. The physician responsible for a given patient's treatment may, however, at any given time be in any other part of a hospital (more often than not). A PDA connected to a hospital's computer network over a wireless connection could provide a physician with a direct link to "his" or "her" patient at any time.

This type of monitoring is exemplified in Figure 2.1. Here, the patient's heart is monitored using 12-lead ECG (described in 6.1), the signals then go from the monitoring equipment to a local area network (LAN), or to the Internet, and from this network to the PDA remotely monitoring the patient's heart.

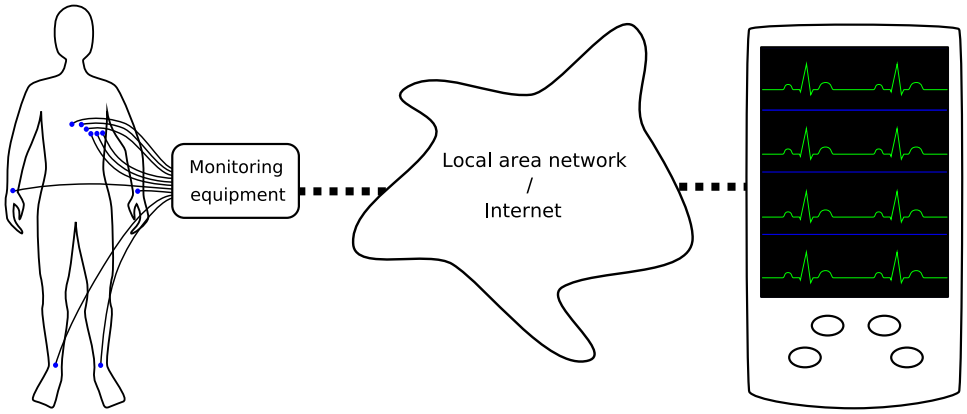


Figure 2.1: Remotely monitoring a patient.

[1]The number of nurses per patient does vary. In Norway, [3] recommends 0.5 to 2 nurses per patient and at least one per shift for unstable patients. Other countries operate with their own specifications.

Chapter 3

Hardware and software alternatives

In this chapter, we focus on some of the more interesting alternatives of hardware and software, and the choices made are outlined.

Preliminary preparations for development of a monitoring application involves making many decisions on several fields, particularly about hardware and software. There is a great diversity of alternatives in both these areas, with different families of PDAs support different operating systems and different programming and scripting languages.

3.1 Network protocol

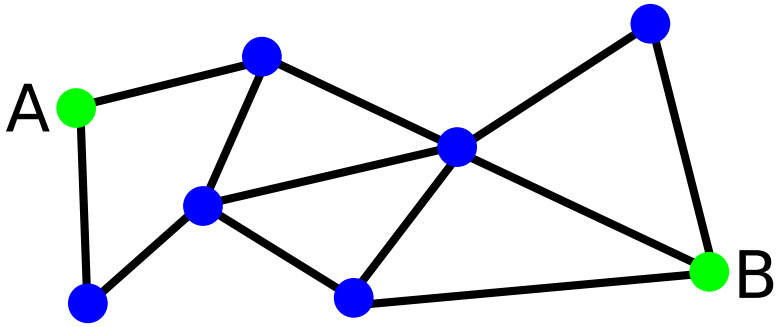


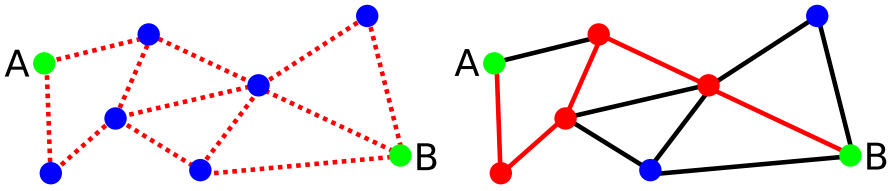
Figure 3.1: A network topology, with node *A* about to communicate with node *B*.

Multiple protocols for data transfer over networks exist, each with different properties. The currently most popular being *User Datagram Protocol* (UDP) and *Transmission Control Protocol* (TCP), both of which work on top of the *Internet Protocol* (IP).

UDP is *connectionless*, meaning that it is not necessary to establish a connection between the sender and the receiver before a data packet is sent and no reply is required from the receiver. In this way, data transfer is efficient but not necessarily reliable:

- A sender sends a data packet but does not know whether the packet arrives at the destination.
- The order of the packages reaching the recipient is not known.
- No packet routes are known in advance.

A network using UDP traffic is exemplified in Figure 3.2(a), where packages can move along any of the routes indicated by the red, dotted lines.



(a) A UDP network topology, with node *A* sending packages to node *B* over either route, indicated by red, dotted lines

(b) A TCP network topology, where node *A* has set up a connection to node *B* over a permanent route, indicated by the red lines

Figure 3.2: The network from Figure 3.1, using UDP and TCP.

The other very popular protocol, TCP, is *connection-oriented*, meaning it establishes a connection between a sender and a receiver before starting to transfer data. TCP was designed to overcome the problem of unreliable *internetworks* (interconnected networks [13]), providing a reliable connection where the medium can not. Wireless networks with much interference are prime

examples of unreliable networks, with much package loss. When a packet either does not arrive at its destination within the expected time, or is corrupted, it is retransmitted.

An example TCP connection, where packages can only move along a pre-setup route, indicated by the red lines, is exemplified in Figure 3.2(b).

Another noteworthy protocol is *Real-time Transport Protocol* — RTP. RTP (normally) transmits its data using UDP and is designed to transmit streaming data, its basic functionality being multiplexing multiple data streams into a single UDP stream. It is in widespread use for e.g. multimedia data transfer, where the real-time aspect is important and there is no need for retransmission of packets as they probably would arrive too late. RTP is also, as opposed to TCP, well suited for multicasting, which is a large field in multimedia streaming.

3.1.1 A UDP / RTP scenario

Depending on choices made when programming the client and server (with the PDA acting as a client, receiving data from a server, see Section 7.3), either bandwidth or reliability can be more important. One possibility is to generate frames on the server and transfer them to the client, as a live video transmission. This would require substantial amounts of bandwidth and little work by the client.

In this scenario, feedback from the client to the server would be more difficult as package loss will arise sooner or later. Although retransmission could be implemented in the application, this would interfere with the layering model of the protocols and could be relatively complicated to implement.

3.1.2 A TCP scenario

Another option than sending finished frames for display, is to transfer measurement data and leave it to the client to present it in a proper manner. This model requires little bandwidth, but can depend fairly heavily on reliable data transfer. If the client requires to communicate with the server, it needs an interactive user

interface, rendering a pure video viewing device relatively limited. With these requirements, and considering that wireless transfer is prone to disturbances from electronic equipment, obstacles etc, a connection-oriented protocol is preferable.

3.1.3 Network protocol conclusion

Although there are several options, some of which are discussed here, and also the option of creating a specialised protocol, TCP stands out as the best and simplest alternative for the following reasons:

- TCP is mature and well tested, being one of the most important networking protocols.
- Many APIs have extensive support for TCP. Implementing network protocols is time consuming and mostly unnecessary as it is often "reinventing the wheel", at least for a proof-of-concept application. Whether this should prove itself worthwhile in the future if the monitoring application should undergo major development is hard to say at the current time.
- Critical data packets are guaranteed to arrive at their destination. While this is entirely possible to circumvent using a connectionless protocol, as long as that protocol does not have required features, implementing such a feature would bloat the code and take unnecessary time from other, more important issues.

3.2 Networking technology

The need for data transferral to the PDA necessitates either a cable or some form of wireless network connection. Having to connect the PDA to a network using a cable in order to transfer data would mean that one would neither be able to see data real-time (when bringing the PDA away from a stationary computer) nor be better off than by using an actual stationary computer with a larger screen, simpler user interfaces, larger computational power etc. This fact, as well as the focus of this thesis on wireless transport, renders the option of using a physical cable relatively

useless. Possible (widely available) wireless alternatives includes IrDA, Bluetooth or IEEE 802.11 (“Wi-Fi”).

3.2.1 IrDA

IrDA — *Infrared Data Association* — is well tested and likely the most widespread wireless technology in the realm of portable devices (including PDAs) offering a decent transfer rate of up to 16 Mb/s. That said, IrDA has a short range (less than one meter) and requires line of sight between the devices that are to connect.

3.2.2 Bluetooth

Bluetooth is a new technology which has reached a significant use. The range — normally 10 meters, but this can be increased by increasing power — is better than with IR and there is no need for line of sight. Bluetooth is also designed to be a low-cost technology but the asymmetric data transfer capacity is limited to 723.2 kb/s with a return rate of 57.6 kb/s, which is fairly limited. Bluetooth is well suited — and in use — for communication between peripherals like keyboard, mice, head phones etc and is finding its way into PDAs as well.

3.2.3 IEEE 802.11

IEEE 802.11 is by the time of writing by far the most popular wireless technology for computer networking. As the technology is more expensive and complex than the two aforementioned, it is normally not found in smaller devices, like mobile phones or peripherals, but few laptop computers are delivered without it and it is also found in some high-end PDAs. IEEE 802.11 is much more versatile than the alternatives as it offers a much larger range and data transfer capacity than either.

3.2.4 Health implications

Although potential health implications caused by exposure to electromagnetic fields have been studied extensively, little research

has been made on the health implications of wireless networking in particular. In a short area around wireless access points, radiation can be notable, depending on the transmitting power of the access point, but there seems to be little scientific evidence to indicate that areas spanned by wireless networks are dangerous to humans. The closest area of research to wireless networking which has been extensively studied is probably mobile phone radiation. Most of this research seem to indicate that, although heating of tissue in close proximity to the transmitting source is reported, the risk of this type of EMF radiation is rather low. Still, the research into these areas is by no means complete, so some caution is generally encouraged [12].

3.2.5 Networking technology conclusion

The purpose of this thesis is to create a proof-of-concept type of application potentially shortening the distance to patient status information for physicians and thus increase efficiency. If IrDA was used, the user would have to stay stationary, and the concept of a roaming patient monitor would not be possible to achieve. IrDA is sufficient for transferring data to a PDA for later study, but not for continuous roaming monitoring.

All the evaluated technologies can support any of the network protocols, so the actual technology to use can to some extent be left to the facility implementing the monitoring application. However, the following observations can be convenient to take into account:

- Wireless technology has been a source for scepticism in hospitals and other areas (like for instance in air planes) for fear that interference from the radio signals should cause vital equipment to malfunction. Lately, some hospitals have opened up for this type of technology and several have IEEE 802.11 wireless networks covering large areas. Where, for some reason, such networks are not available, Bluetooth can be an alternative as the range is normally shorter, but this decision must be made individually, in each implementation case.

- IEEE 802.11 networks have a larger range than Bluetooth networks, and the data transferral rate is larger. This allows

for fewer access points and reduces the chance of congestion in the network.

- It would seem that the issue of health implications caused by the radio waves is negligible (see Section 3.2.4), though new research can prove otherwise. If this should be the case, this is likely to affect which technologies are available in which areas so some flexibility with regard to the exact technology to use is recommended.

3.3 Operating system

There are three major operating systems in use on PDAs: Microsoft Windows, Palm OS and Linux. They all provide both advantages and disadvantages, related to ease of use, stability, development cost and more. We briefly and informally discuss some of them here.

3.3.1 Palm OS

For a long time, Palm PDAs have been the most popular and Palm OS is probably among the most stable OSs for PDAs as it was designed from the start to be an embedded OS and has had more years of active development than its competitors. Palm OS's API is freely available along with other documentation and development tools are free as long as one registers[26]. On the other side, Palm OS is proprietary, meaning one is in practise limited to the API provided by Palm, and the programming tools are platform dependent, requiring the programmer to use a Windows or Mac computer for development[1].

3.3.2 Windows

The Windows OS available for PDAs is labelled *Windows CE*, *Pocket PC* or *Windows Mobile*, the latter having taken over for the former

[1] In the latest version of WxWidgets, there is alpha support for Palm OS, meaning that one can program for Palm using WxWidgets on a large variety of platforms. Being in alpha state, however, this library is not well tested yet.

as the Windows line of operating systems for PDAs. Windows is a well tested, market leading OS on the x86 PC hardware platform, but on the PDA platform its market position is rivalled by Palm OS. The Windows operating systems comprise proprietary, mostly closed source, with the result that many of the development tools and other software are limited to that platform and can be costly. On the other hand, the availability of software is vast and it is unlikely to be a poor choice of development platform. There is also a greater variety of possible programming languages one can use than with Palm OS.

3.3.3 Linux

A relative newcomer in the field of PDA operating systems, Linux may not be as stable as its competitors. Apart from its young age, the fact that it is largely maintained and developed on a voluntary basis makes it more vulnerable with regard to the resources invested in it in fields where the user base is small. On the upside, in areas with larger user bases, Linux thrives and develops rapidly, and coupled with the fact that most software for Linux is freely available, this makes it at least an interesting choice. Programming tools are freely available, more programming languages are available than with any of the other OSs, and one is free to work on any platform.

There are currently several Linux distributions tailored for PDAs, most based on the *Debian* system. Two of the larger are *Familiar Linux* for the iPAQ family of PDAs made by *Compaq* and *OpenZaurus* for Sharp's Zaurus PDAs.

3.3.4 Operating system conclusion

Among the various operating systems targeting embedded devices, Linux has recently emerged as the OS on which most development is taking place [6]. The reasons are many, but among them the need for a "common ground" for development. There exist many operating systems for embedded systems (most of these are not for PDAs), meaning that all necessary software must be developed for each of them, which is a huge and expensive task.

Many common libraries on Linux are released under the *Gnu Public*

License (GPL). This licence states that other software incorporating GPL licenced software must also be licensed under the GPL, and enforces openness of the software; you can commercialise the software, but you always need to provide the source code without charge ensuring openness and facilitating reuse. The “viral” nature of this license has lead to the development of a plethora of open source software available to all, which means that developers can focus on the task they need done, instead of reinventing the same software over and over again.

Many more microprocessors and microcontrollers go into embedded systems than into workstations, so the application area for operating systems is greater. As Linux surpasses established competitors in volume it proves it is a force to be reckoned with. This, in conjunction with the vast amount of tools available, makes it a choice for development that goes beyond personal preference.

The above reasons were important in the decision process, but personal preference was also important. Especially in the development of a proof-of-concept application, use of familiar tools is normally a time saving advantage that is not easily matched by less familiar alternatives, and Linux was therefore chosen.

3.4 Programming language

An important factor when choosing the programming language for a particular task is personal preference. Naturally, it is easier to see how programming a specific project can be accomplished with a tool that one knows well than with a tool one doesn’t. It is therefore all the more important to evaluate the tasks and challenges posed in a project with regard to the tool to be used.

Some of the most widespread programming languages for use on PDAs are C, C++, Java and Python.

3.4.1 C

C is one of the oldest conventional programming languages in daily use all over the world and remains one of the most popular. C gives the programmer very detailed control over the actions of a software program opening for very efficient programs, but this also means

that a programmer to a large extent must know very well what he or she is doing. For example, in C programming, it is necessary to specifically allocate memory space for data and free this after use, a task which many C programs do poorly.

C probably has more APIs than any other language[2] and is therefore a "safe" choice when you are in need of functionality in a small field.

3.4.2 C++

A successor of C, *C++* was created to be an *object-oriented* extension of the former[11]. C++ includes almost all of C as a subset and can thus give the programmer the same fine grained control over the computer. However, C++ also provides a more high-level API than C, easing tasks like memory management. As programming C++ in practise also relies heavily on the use of the so-called *standard library* or similar libraries, one can create large C++ programs without the deep knowledge required of a skilled C programmer.

3.4.3 Java

A modern, fully object oriented programming language, *Java* aims to be almost totally platform independent, enabling the use of the exact same program on different hardware platforms and under different operating systems. This is of course a huge advantage, but in order to accomplish this, one uses a *virtual machine* (VM) to execute Java programs, as opposed to the former languages' programs, which are executed natively on their respective platforms. Although there is constant work going on in order to lighten the load of this extra layer, Java programs are often more memory and CPU intensive than natively compiled programs. The huge amount of generality provided by Java is also to some extent traded for loss of the fine grained control the others provide.

[2]Many APIs of other programming languages, as well as their compilers and interpreters, are actually written in C.

3.4.4 Python

The only scripting language among the languages mentioned, *Python* is both a popular and relatively efficient language. It is a very high-level object-oriented language, and thus quick to program. The fact that it is a scripting language also means that it is not compiled, like C and C++, but interpreted. Somewhat similar to the VM of Java, the Python interpreter also takes up memory and CPU power slowing the programs down.

3.4.5 LabVIEW

The system created in [8] and briefly described in Section 2.2.1 was written using *LabVIEW*. LabVIEW is a graphical programming language that specialises in measurement and laboratory data display. It is a powerful language with much built-in functionality for many or most of the tasks necessary in this thesis. In May 2003, a PDA module was introduced, currently running on several Windows PDA versions and Palm OS. LabVIEW requires a development program, only available from its creators, *National Instruments*.

3.4.6 Language conclusion

When a medical application is to be developed, LabVIEW should be considered an alternative. At the onset of this thesis, however, the PDA version was not yet available, although it soon became so. As LabVIEW is not a general purpose programming language, making potential limitations hard to overcome, combined with the need for specialised software, limited knowledge about the language and a wish for independence from a proprietary system, it was discarded as an alternative for this proof-of-concept application.

As the PDA has small amounts of both CPU and memory, and taken into account that the monitoring application needs to run in real-time, there is a large need for efficiency. A certain degree of control over low-level functionality might also be practical. Since both Java and Python provide less of all of this, C and C++ stand out as the best alternatives. Furthermore, in accordance with modern code design principles, modularity and ease of reuse

and extensibility are among important issues to consider. Object oriented programming provides a basis for this. As there also exist APIs that simplify C++ programming as opposed to pure C programming, C++ seems the best choice.

3.5 Windowing toolkit

There exists many different windowing toolkits, and abstractions over these as well.

- The *Gimp ToolKit — GTK* was originally made for the image manipulation program *Gimp* and was later adopted as the toolkit used for *GNOME* development[24][3]. Although GTK is not platform specific, its use is greatest on Linux. Larger programming frameworks again can work as an abstraction over GTK and use this (or other windowing toolkits) for Linux compatibility.
- *WxWidgets* (formerly WxWindows), can use either GTK, *Motif* or *X11* as windowing toolkit for Linux compatibility, or other toolkits on other platforms.
- Windows uses its own toolkit, but other toolkits have also been made available, like GTK.
- Like Windows, Mac OS has its own, platform specific, toolkit, but also here, others have been ported or developed for Mac.
- The portable *MGL* graphics library.
- *OS/2* has its own toolkit.
- Palm OS has its own toolkit, and WxWidgets has recently been ported to Palm.
- Another cross-platform development library, available for Windows, Linux and Mac OS is *QT*. QT is not an abstraction over other windowing toolkits as WxWidgets can be, but works directly with X11 on Linux or the other OS's integrated toolkits. QT has also been ported to embedded systems and there been labelled *QPE*, and later developed further under the name *QTopia*.

[3] GNOME is one of the major Linux desktop environments.

With the three available OSs for PDAs, there are many toolkits to choose from, but Linux offers the greatest range. Two of the major embedded Linux distributions, Familiar Linux and OpenZaurus, both support the desktop environment *OPIE* (Open Palmtop Integrated Environment), and Familiar Linux also supports the *GPE* (GPE Palmtop Environment, originally GNU PDA Environment[21]).

GPE is very similar in functionality to a desktop environment like GNOME, and it uses the GTK+ windowing toolkit and the X11 X server. OPIE is made from a branch of QTopia and development is thus done using the QT library. OPIE does not use an X server, but writes directly to the PDA's *framebuffer*[4], making it more lightweight than the alternative as X server operations are expensive with regard to memory and CPU use.

As Linux was chosen as operating system, the windowing toolkit alternatives were limited to GPE and OPIE (one can also omit a user interface, by installing a *bootstrap* installation, see Section 4.3.2). GPE (potentially) requires more resources than OPIE and OPIE code is written i C++ using the QT API, making it more portable. Although not initially an obvious choice, OPIE soon became the preferred development environment and windowing toolkit for these reasons.

3.6 Hardware device

Electronic hardware is in rapid development, with exponential growth in many areas. A choice made one or two years ago may not seem as relevant today as it may already be outdated and whole technologies can have vanished and been replaced. The number of usable PDAs is greater at the current time than it was at the onset of this thesis over a year ago.

Among the large diversity of different PDAs available, only some were suitable (and available) for the development of the monitoring application. The most important criteria is (in no particular order):

- Size. The screen should not be too small, as this would give little room for data presentation. The PDA itself must also be

[4]The framebuffer is the part of the computer memory where graphic frames are stored before or as they are displayed on a screen[17].

sufficiently small that it may fit in pockets or be convenient to bring along without being intrusive. Most PDAs strive to meet this criterion, so this does not rule out many models.

- Wireless networking. The intention of the monitoring application is to work as a quickly available and portable source of information for medical personnel, necessitating some form of wireless networking.

- Computational power. In order for an application to continuously display data, the device on which it runs needs to satisfy some minimal level of computational abilities. Most, if not all, modern PDAs should be able to do this satisfyingly.

- Memory. A minimum amount of memory is also required, though, as with computational power, this requirement is also met by most PDAs.

- Support for the desired operating system. In the case of this thesis, Linux was the preferred choice, so this limits the amount of available PDAs.

Needless to say, more memory and computational power is a good thing, but as the development in these fields progresses at a fast pace. Given the safety requirements imposed on computerised aids by hospital regulations, before any application can be put into use in practise, current hardware is likely to be outdated. At the time of the onset of this thesis, few PDAs satisfied all the above criteria to a satisfactory extent, but at the time of writing, over a year later, many models exist.

One PDA family stood out as the best alternative for development, namely the Hewlett Packard (HP) / Compaq iPAQ series. They are relatively state-of-the-art with regard to memory and computational power, and HP has supported a research project where Linux has been ported to this hardware platform. They were at the onset of this thesis among the few PDAs with built-in support for the IEEE 802.11 wireless network technology. Due to availability, the final choice fell on the HP iPAQ H5450.

3.6.1 The future of PDAs

Predicting the future is, needless to say, an impossible task. The development of mobile telephones have created some scepticism towards the future for PDAs as much of the functionality covered by PDAs has been incorporated into telephones and some telephones look more like hybrids comprising telephone and PDA. Despite this, the first quarter of 2005 saw a record in PDA sales worldwide and as more operating systems and software become available, the areas of usability for PDAs seem to expand.

Whether PDAs will continue to exist or not, portable devices with networking technology are not likely to disappear. Neither is it probable that the current rate of increase in computational power and memory and storage capabilities should come to a halt in the foreseeable future.

Future development in this area is likely to present new possibilities for roaming patient monitoring; a hybrid telephone / PDA can receive data using for example the mobile data service GPRS without the need of an intermediate computer network. Telephones with this ability are available today, and a future common ground for development can facilitate portability between devices. The Linux operating system has been gaining ground in the embedded market[6] and can thus provide such a common ground.

3.7 Hardware and software summary

The combination of hardware and software that was finally chosen was guided by several factors, the most important of which was available hardware. The choice of operating system was also an important factor, and personal preference was given much weight here.

As Linux was the preferred choice, a HP iPAQ PDA was already the preferred hardware device (because of its Linux support) before I had access to a test unit, and when I had this access, the choice was quickly settled.

Choosing Linux as operating system ruled out Windows and Palm OS specific software, so for the windowing system, three alternatives remained that could be run under the Familiar Linux

distribution, namely GPE, OPIE and the bootstrap installation (described in Section 4.3.2). The latter implied too much work to set up with a working graphical user interface to be any real alternative, as this would have overshadowed the application development in time. GPE was a good alternative, but as OPIE provided both a well tested and easy-to-use API and also direct access to the framebuffer, omitting memory and CPU expensive X operations, OPIE came out the best choice. OPIE software can also be used on OpenZaurus and can easily be rewritten to work with QTopia, further increasing the number of devices it can be used on.

The choice of programming language was already leaning towards C++, but selecting OPIE as windowing toolkit ruled out other languages.

Selection of network protocol was not guided mainly by hardware or software, but by the need for the monitoring application to have a reliable connection. TCP/IP provides this and is also well tested. Furthermore, it is widely implemented and supported, facilitating development.

The last alternative discussed was networking technology. The only “real” alternatives were Bluetooth and IEEE 802.11, as IrDA is not suitable for roaming devices. Between the two remaining, no real choice was made, as both support the TCP/IP protocol and can therefore be used. However, the application was developed and tested using IEEE 802.11 as Familiar Linux did not support the Bluetooth hardware in the PDA model used.

Chapter 4

Embedded Linux

On a workstation, storage capacity, memory and computational resources are, if not limitless, vast in comparison with embedded devices, which can be limited to only a few megabytes of storage and memory (and sometimes even less). Often, the CPUs also lack functionality found in general purpose computers, such as dedicated floating point processing hardware.

Linux did not originally target embedded systems, but was designed for the workstation / server segment. During its lifespan, however, Linux has been further developed by an ever increasing number of developers and is now ported to many hardware platforms, like *x86*, *Alpha*, *Sparc*, *MIPS*, *SuperH*, *PowerPC* and *ARM*[10].

Operating systems running on embedded systems need to have a small memory footprint and can not require the same computational resources as when running on workstations. Another important issue is robustness, where demands on embedded systems are higher. When the power on a PC is switched off, it normally goes through a shutdown procedure where files are written to and closed, processes are stopped in a controlled manner and contact with other computers is closed down. Until this process has finished, the power isn't switched off. Failure to complete such a procedure may result in data corruption and loss, but as PCs normally have stable power connections, this is generally not a big problem. On embedded devices, this stable supply of power may be replaced by short lived batteries which can suddenly be emptied. The device may also have to be shut down quickly (e.g., turning off a mobile phone in a meeting or in a movie theatre) or it might shut down

unexpectedly for other reasons. If the device is shut down while writing to disk, data loss might ensue or system critical files could be corrupted, rendering the device useless or the system in need of restoration.

Many previously established embedded operating systems were designed specifically for this environment from the start, providing them with performance and stability advantages that are difficult to match for general purpose OSs. Nevertheless, the interest in adapting Linux to this segment has been great for a number of reasons, some of which are:

- There is no licence fee for using Linux.
- As Linux is open source software, the source code is freely available, enabling developers to tailor the OS to fit their applications in ways not available with closed source systems. This also eases debugging, as the whole system is transparent.
- Linux is generally renowned for being very stable.

4.1 Boot sequence of Linux systems

For any computer to load an operating system, it must be told which code to execute and how. On PCs, this is handled by a special sector on the hard drive (or other bootable media), called the *master boot record* (MBR). The *BIOS*[1] of the computer instructs it to load the MBR, and the MBR in turn loads the operating system.

The MBR will load the Linux kernel and also often an *initrd* — **init**ial **r**am**d**isk[14] — into memory. This initrd contains a miniature Linux version and its job is to enable the loading of the main OS, in particular, it contains hardware drivers that are not in the kernel and are necessary for, e.g., mounting the system hard drive partition. After the necessary drivers are loaded, the file systems are mounted, the initrd is swapped for the main OS and the system comes up.

[1] The BIOS, or Basic Input/Output System, is the first software run when a computer is switched on. The BIOS in turn loads whichever system, if any, that is set up to be loaded, normally from a hard drive, but also from other media, like cd-roms or floppy disks.

This procedure is very similar on embedded devices, but there is normally no MBR as in PCs. On the iPAQ, there is a boot block, which loads a boot loader, both residing in the flash memory. The boot loaders are hardware specific and can only load specified operating systems. For example, for any embedded platform, a boot loader supporting Linux must be available for Linux to be able to boot.

The initrd exists for loading drivers necessary for booting a system, but these drivers can also be compiled into the kernel. If all possible drivers were compiled into a kernel, however, it would become prohibitively large and take up too much of a system's resources. The ideal solution is to compile a kernel specifically for a given hardware configuration, but many workstation users do not want to or can not for some reason compile their own kernels. To be able to use the same kernel in many different systems, an initrd can load the necessary drivers not included in a general purpose kernel. On embedded systems, the initrd step would normally be skipped, to save space and make a streamlined kernel, improving performance.

4.2 File systems

Many different types of file systems exist. The purpose of file systems is to store data and provide an interface to it, and they accomplish this in different ways.

Some of the more common file systems on Linux systems are Ext2, Ext3 and Reiserfs. Ext2 was the main file system used in Linux systems until recently, and is still in use in many areas. Ext2 is not a good choice for embedded devices like PDAs, as it is not very secure against data loss, to which these devices are very vulnerable.

The successor of Ext2, Ext3, as well as the Reiserfs file system, includes an important feature for embedded systems, namely *journaling*[20]. A journaling file system prevents errors caused by power failures and system crashes by keeping a journal of all its actions and writing to this journal before it performs the action in question. An example is deleting a file on Linux file systems, an operation that is performed in two steps (example from [20]):

- First, the file's directory entry is removed.

- Afterwards, the space occupied by the file is marked as free in the free space map of the file system.

If either of these steps is not performed because of some interruption, it leads to file system inconsistency, either with the file's directory entry still existing but the space it is occupying marked as free (and thus over-writable), or the file's directory entry being removed but the space not marked as free (resulting in loss of storage capacity). When the system keeps a journal, all it needs to do when it comes back up is see whether it was able to accomplish all it was supposed to do before the system went down and then quickly clean up after itself.

VFAT is an extended version of the DOS file system, and is used by Windows 95 and Windows NT. This file system is also the one most used on memory extension cards that are often used with embedded devices, as Windows is the dominant workstation operating system and most other systems can read VFAT.

4.2.1 A Flash memory file system

Embedded devices must regularly endure harder conditions than workstations, for example being shaken and dropped. Flash memory has no moving parts and is better suited for these conditions, and it is therefore the dominant storage medium used in embedded devices. A special characteristic about Flash is worth noting: In Flash memory, erasing a word requires erasing a whole *erase block*, a sector, which on the iPAQ is 256kB. Each erase block can only be erased about 100000 times[7].

This means that frequent writes to the same sectors of Flash memory will result in early failure of the memory. To avoid this, one can use a *wear levelling* system, either as a layer under file systems like Ext3 or VFAT, or use a file system that incorporates this. JFFS2[19] (Journaling Flash File System) is such a system, in many ways similar to other Linux file systems, it is designed specifically for Flash memory and incorporates journaling and wear levelling.

4.3 Running Linux on the HP iPAQ H5450

The iPAQ family of PDAs was, and still is, designed to work with the Windows operating system. Because Compaq and HP ran a research project (see the Preface), and because of the support of many individuals, Linux can now run on many iPAQ PDAs.

In general, large obstacles present themselves when an operating system is to be ported to a new hardware platform. A complete hardware platform consists of many components and their microcontrollers working together, such as:

- A hard disk drive or some other type of data storage system. PDAs generally use flash memory for this.
- Memory unit. In the iPAQ H5450, there is 64MB of DRAM, which is erased if the battery is depleted. This memory is used both for user data storage and for memory.
- Monitor / screen controller. The computer needs to communicate with the user in some way, most do this via a monitor or screen.
- Network controller. To be able to communicate with other computers, a network controller is necessary.
- User input device controllers, for keyboards, mice, touch screens and other means of providing input to a computer.
- Various device controllers. These devices include Universal Serial Bus (USB), serial and parallel buses, sound controllers and many others.

Some or all of these are hardware components comprise PCs, PDAs, and other types of computers. Controllers are manufactured by different companies and usually the computer producers buy their components from these manufacturers and assemble them into their finished products.

Few internationally acknowledged standards exist that specify how these components communicate with each other, so most manufacturers make their own specifications. For a system to communicate with a given component, a *driver* for this component must be made, and for a driver to utilise the full potential in a given

controller, the specifications of the controller must be known to the driver programmer. One example is graphics cards, which often contain dedicated graphic hardware designed for 3D acceleration — improving performance of three dimensional imagery.

Most hardware component manufacturers keep specifications of these components to themselves, making it difficult for others to make drivers for other operating systems than the ones the manufacturers intended them for. More often than not, the operating system this hardware is designed for is limited to some Microsoft Windows edition, and for Linux and other OSs, drivers for this hardware need to be written "blindfolded", that is, without access to the specifications of the component in question. This process is called *reverse engineering*, as it involves starting with a finished product, often a hardware driver, and by looking at that, trying to establish how it was written. Implementing all functionality of a component is very difficult in this way, with the result that specific functions of specific components often are inaccessible to other operating systems than the one or ones intended by the manufacturer.

All of this makes Linux support for specific devices somewhat haphazard. Some manufacturers do release Linux drivers for their hardware and others are made by people either working for companies requiring Linux support for specific hardware or by people in the open source community.

At the time of writing, all the hardware on the iPAQ H5450 is supported by Linux to some extent, except the Bluetooth interface, so although Bluetooth can be used with the monitoring application, it has not been tested. The IEEE 802.11 interface is supported, however, and the application works fine with that.

The iPAQ can be used with a *cradle*, a docking station which charges it and can be connected to a workstation using either a serial or a USB cable. During work with the iPAQ, I used both types of cables. On the iPAQ H5450, a workstation can only communicate with the boot loader using a serial connection, and for other purposes, a USB connection is faster.

After installation, I first used the terminal connection and set up a Point to Point (PPP) network connection to the workstation. Then, the workstation was configured to route the traffic from the iPAQ onto the Internet. After establishing an Internet connection, software packages could be downloaded and installed

automatically. The role of the terminal PPP connection was later taken over by a USB connection, which increased the data transfer speed and facilitated development. Later, when a wireless network access point was available, this further facilitated networking as the iPAQ did no longer have to sit in its cradle for connectivity.

Because the storage space of the iPAQ is strictly limited, I used a network share from a workstation and mounted it using NFS, essentially giving the iPAQ several gigabytes of storage capacity. This, although feasible with a terminal PPP connection, is much more efficient with a USB or a wireless network connection.

4.3.1 Preparations for installation

Before installing Linux on the iPAQ H5450, [7] was consulted. For communicating with the PDA's boot loader, the software that loads the operating system, a terminal connection to a workstation was set up using a terminal emulator that also handles file transfers.

The pre-installed Windows CE was backed up using the pre-installed boot loader, then the boot loader itself was replaced, as the default boot loader can not run other operating systems than Windows CE. This replacement boot loader was originally developed at Compaq's CRL and later further developed by the open source community at handhelds.org.

4.3.2 Installing Linux

To install Linux, a pre-compiled installation image was acquired. As this procedure has been repeated several times, different images have been tested:

- A bootstrap image containing no user interaction software. This can be used for remote access, development and testing, and any kind of desired user interaction software can be set up.
- An OPIE image. This is one of the two pre-setup environments for Familiar Linux. OPIE was later selected as the environment to use for development.

- A GPE image. The other of the two pre-setup environments for Familiar Linux. Both OPIE and GPE are studied in Section 3.5.

All the above images can be downloaded from the Internet or built using the OpenEmbedded build system described in Section 5.3.2. The images are transferred to the PDA using the terminal emulation software and installed using the specialised boot loader. After this procedure is complete, one has obtained a running Linux version and can continue to install or upgrade software packages over a network connection using Familiar Linux' packaging system, *ipkg*.

As the OPIE distribution was not originally a clear choice for development platform, all of the above systems were tried, in different configurations. The bootstrap image is a good place to start to tailor the system entirely after own specifications and I experimented with setting up an X server and different window managers, like *fvwm2* and *matchbox*. I also tried different ways of compiling the software (see Chapter 5).

4.3.3 Setting up multiple systems with the monitoring application

Setting up the patient monitoring application on a PDA, either without an operating system or with a different operating system than Familiar Linux with OPIE, requires a tedious procedure of acquiring an OPIE installation image, either from the Internet or by building one using OpenEmbedded, installing the image and after upgrading the software packages that need upgrading and setting up the system, installing the application.

The system setup would normally include setting up a networking connection, increasing the security of the device by adding a root password (the default is blank, which can cause a major breach of security), setting the time zone and possibly installing and setting up other software packages as needed.

When upgrading packages, newer versions of libraries can cause problems, particularly as there is a high degree of development on most of the software in these distributions. These problems can arise if libraries change their functionality. Upgrading the

system therefore necessitates rigorous testing, and this can be time consuming if repeated too often.

For deploying the monitoring application on a larger scale, one would follow the above steps to create a working installation. Afterwards, one would use the boot loader to take a *snapshot* of the installation, creating an *image*, a file containing the whole file system. This image can then be used to create exact copies, again using the boot loader, of the working system for mass production.

By producing an image in this way, and by utilising technologies like DHCP, for automating network configuration, and NTP, for automating time setup, the process of installing a PDA with the monitoring application can be reduced to two steps: installing a boot loader and then installing the image.

Chapter 5

Software compilation

Software compilation is the process of making either *machine code* or *byte code* from *source code*. Source code is the program as written by a programmer. Machine code and byte code are unreadable to most people, but while machine code is executable by a computer, byte code is in a kind of intermediate state, not being executable directly by a computer, but requiring a platform specific program to run.

Interpreted programming languages, or *scripting languages*, are used to write programs that are interpreted when they are executed. This means that the program code is not transformed in any way before execution and that the code is not evaluated before run-time, at which time it is parsed and transformed into a machine-executable form by an *interpreter*.

Compiled programming languages need to be compiled before run-time. This means that the procedure of transforming the program code into machine-executable form is done before run-time, enabling a *compiler* to find errors in the program before it is executed. Compiled programs are also generally faster as they are already parsed and machine-executable.

Byte code compiled languages are compiled into byte code, which is not directly readable by a machine, but by a *virtual machine*, a program that needs to be written and compiled specifically for each platform it is to run on. Byte code compiled languages are generally slower to execute than purely compiled languages, but their strength lies in platform independence, as the same byte code can run on all platforms that have a virtual machine for the given language.

If high efficiency or a high degree of control over the computer is desirable, compiled languages is normally the natural choice. A major drawback with this type of languages is that they need to be compiled explicitly for each platform they are to run on. To achieve this, there are two alternatives: *native compilation* and *cross compilation*

5.1 Native compilation

Compiling natively means compiling on the platform on which the software is to run. This is the usual way to build (compile) a program and by far the easiest, as practically every piece of software uses platform specific libraries. This means that all libraries used must be compiled and likely modified for each platform the software is to be compiled on. When compiling natively, most such libraries normally already exist on the computer, as most libraries are “common” libraries used by a large amount of software. This includes software used on the computer on a daily basis, so normally, there is little effort needed for setting up a library environment for native compilation.

5.2 Cross compilation

In cross compiling, two different platforms are involved, the *build platform*, where the software is compiled, and the *target platform*, where the compiled software is to run. The tools used for a compilation are often referred to as a compilation *toolchain*, which normally comprises the libraries used, the compiler, the *linker* (which links the program to the libraries), and a collection of other, smaller tools. The toolchain must be configured and compiled to produce machine code for the target platform, while running on the build platform. To achieve this, one must start with a compiler that supports cross compiling (possibly build this first), and use this compiler to build the rest of a cross compilation toolchain. Then, this toolchain (including the cross compiler) can be used to build the software that one wants to compile (see Figure 5.1).

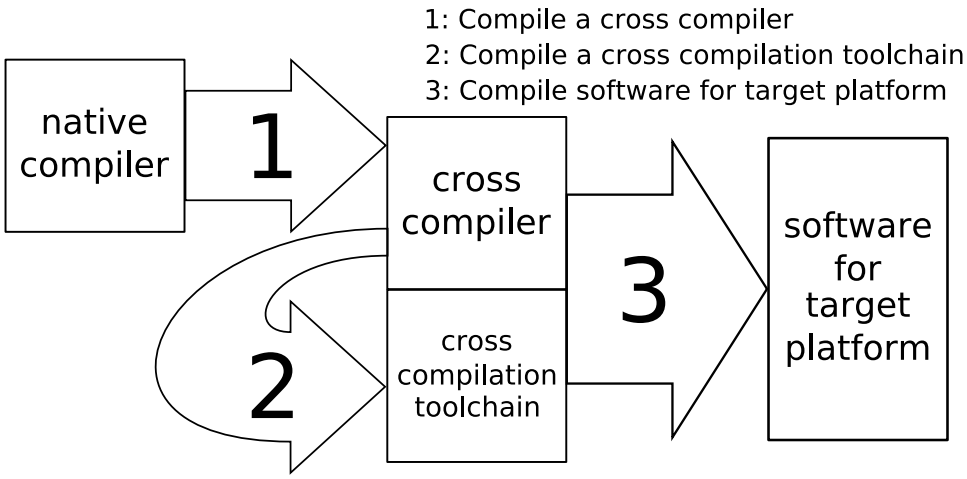


Figure 5.1: Building a cross compilation toolchain.

5.3 Compiling software for embedded devices

When compiling software on a workstation, native compilation is normally the natural choice. On embedded devices, computational power, memory and storage are generally sparse resources, often making native compilation difficult, primarily because the storage space is so limited, but also because computational power and memory are sparse resources.

5.3.1 Compiling natively on embedded devices

Most embedded device have prohibitively small amounts of storage and memory to compile natively, but many modern PDAs can achieve this. For compiling natively on Familiar Linux on an HP iPAQ H5450, there are two relatively simple measures one could take in order to provide storage space for a compiler and libraries:

- Use an NFS mounted file system. When the iPAQ is connected to a network, a workstation on this network could provide an NFS share, enabling the iPAQ to use this file system as its own. In this way, the fairly limited 32MB flash storage originally availably can be raised to several gigabytes or even more.
- Use a flash memory expansion card. For the iPAQ H5450, the available type is MMC (Multimedia Card), but other PDAs

often support other types, like Compact Flash (CF) or Secure Digital (SD) cards. These cards come with up to several gigabytes of space, providing ample storage space for a compiler and libraries.

While these solutions solves the storage issue, neither the computing power, nor the memory issue, are affected. With as little RAM available as 32MB, a compilation, at least incorporating a few libraries, can easily run out of memory and will be very slow.

Another, viable, alternative for native compilation on the iPAQ is to use the iPAQ / skiff cluster made available by Compaq CRL. This is a publicly available compilation cluster where anybody can log in using telnet or ssh, and compile the software they wish. The PDAs in this cluster have extended storage capacity and some extra memory. At the time of writing, however, the cluster is down and due to HP's closure of CRL (see the Preface), when or if it will be back seems uncertain. A photo of the cluster can be seen in figure 5.2.

5.3.2 Cross compiling for embedded devices

While it is entirely possible to set up a cross compilation toolchain oneself, there are several available that are already correctly configured and compiled with ARM as target platform. The Debian Linux distribution has one available for installation through its package managing system, *apt*, there is one available on the website http://www.handhelds.org/, and multiple others are also available in different places.

The main advantage of using a precompiled toolchain is that there are often minor modifications to the code of the compiler or core libraries that must be made, requiring a high degree of knowledge of both programming language and the target platform. Another alternative is to follow guidelines set up by somebody with such knowledge and compile a toolchain oneself.

Both these alternatives are possible ways of setting up a cross compilation toolchain, but they still leave important issues untouched. Package dependencies (packages requiring that other are installed) can recurse and become time consuming obstacles, and many packages are not particularly well suited for cross

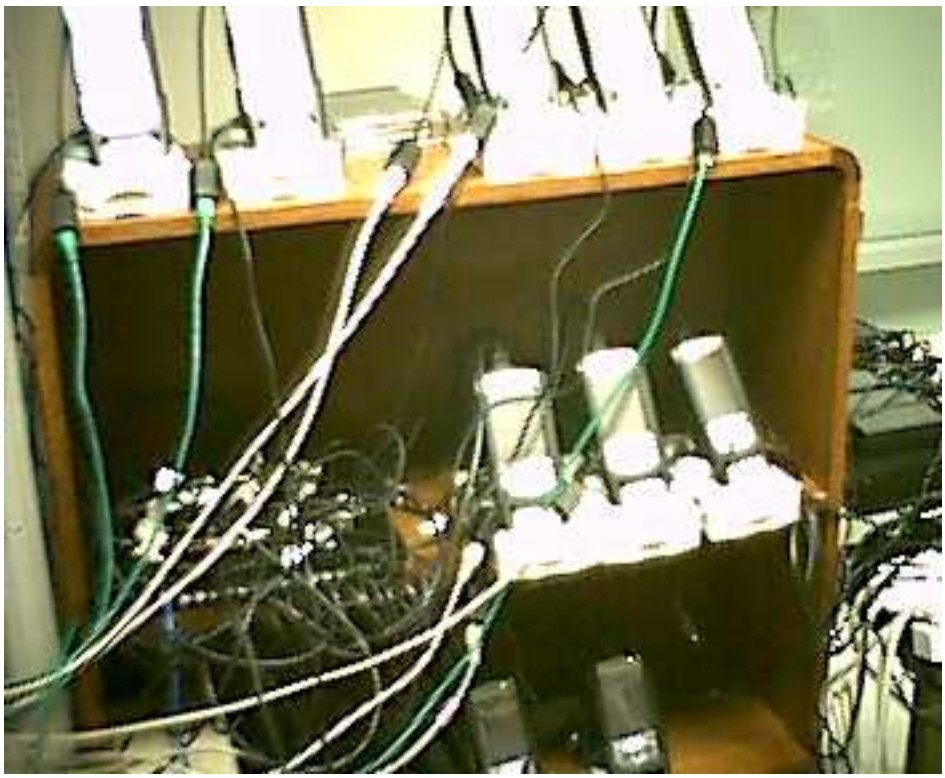


Figure 5.2: iPAQ / Skiff cluster. Image from http://www.handhelds.org/cam.html.

compilation, requiring code patching or modifications made to the packages' build systems.

A third alternative is to use the *OpenEmbedded build system*[22]. This is a set of tools and recipes for building different software packages, including a system for incorporating new software packages. After being set up with the desired target platform, Linux distribution and package format, OpenEmbedded uses the *BitBake* program to compile software and building a specific software package is generally as simple as

```
$ bitbake <package name>
```

Then, if the package in question has dependencies, these are built (recursively) in the right order, and finally, the requested package is built. Building all packages in the package repository at any given time, is as simple as

```
$ bitbake world
```

and building for example a complete OPIE version from all the latest sources is accomplished by

```
$ bitbake opie-image
```

This makes OpenEmbedded a tool that greatly facilitates cross compilation. Some of the goals, most of which are more or less achieved, of the OpenEmbedded project, as stated in [9], are:

- **Handle cross compilation**. This is necessarily an indisputable demand for a build system that does not compile natively.
- **Handle inter-package dependencies**. If one package depends on another, this is recorded in the build file for that package. In this way, packages required by others are compiled first.
- **Be Linux distribution and architecture agnostic**. OpenEmbedded targets support of most, if not all, Linux distributions available for different hardware architectures, making it a very versatile system.
- **Support multiple build and target operating systems**. Although Linux is currently the main operating system in use with the OpenEmbedded build system, other OSs can also be used, both as build and target platforms. Work is in progress to make the build system work on FreeBSD and Mac OS X.

A system like OpenEmbedded presents a powerful toolkit more or less independent of Linux distribution and architecture that greatly simplifies the task of software compilation for embedded systems. The focus for developers can thus be taken away from setting up a build environment and making alterations to software packages to make them work for their target platforms, and enable them to concentrate on the actual development of their own applications.

5.4 Compilation experiences

When first starting developing for the ARM platform, I set up the toolchain provided by the handhelds.org community. After the toolchain was set up, it soon became apparent that compiling the

required libraries was a big task as the dependencies of central libraries recursed into a large number of other libraries. As a consequence of this, I changed to using the iPAQ cluster described in Section 5.3.1, and successfully compiled several packages there.

When compiling on the iPAQ cluster, I only used it to set up software for running the base system (and the common libraries it needed). The monitoring application itself was not compiled on the cluster, since the cluster has many users and is relatively slow to work on. When I got to the development phase, I set up a native compiler using an NFS mounted disk. Although this worked, compilation of necessary, larger libraries depleted the PDAs memory and a different solution was required, and the OpenEmbedded build system (described in Section 5.3.2) became the next compilation tool used.

Setting up OpenEmbedded proved itself to be a quick and painless procedure, which was well documented. The BitBake build tool uses a well documented file format for its building recipes, making it easy to add new packages. Compilation of existing packages was also very easy, and a complete OPIE distribution was made using this tool. BitBake supports QTs build system *qmake* and also the GNU *autotools* build tools. For compiling the monitoring application, qmake was used and for compiling the benchmarking applications, the GNU autotools was used.

To summarise, all these compilation options were viable, but both a “traditional” cross compilation toolchain as well as native compilation over NFS can be hard to work with. The skiff / iPAQ cluster is a good alternative, but can be a little slow to work with, it can not be guaranteed to be online, and using it provides others with access to your code (which one may or may not wish). In my experience, OpenEmbedded is by far the best solution, as it can run on most workstations, enabling it to be highly efficient, and the build process of libraries not currently installed is much easier than with any of the alternatives.

Chapter 6

Automated pattern recognition in ECG signals

Thus far, the field of data processing that has been studied is data *transportation*, how to transmit the data from the source to the PDA (see sections 3.1 and 3.2). The field to be discussed in this chapter is data *interpretation*, that is, how vital and meaningful information automatically can be extracted from the data that is received.

For a physician, access to data requires some kind of presentation in a format he or she can interpret and extract meaningful information from. However, this interpretation process can, in some areas, be taken over, fully or in part, by a computer.

In the case of *ElectroCardioGram* (ECG) signals, a trained cardiologist (a physician specialising in heart diseases) can extract large amounts of information from looking at an ECG plot, but although this can provide him or her with an intuitive idea of, for example, the heartbeat frequency, the exact frequency is difficult to determine in this way. This type of work is ideal for a computer, but it is not straightforward.

There are several different measuring instruments providing plottable data, like measurements of respiration, blood pressure and ECG. I have chosen to focus on ECG signals, as these are among the more important ones (of course, this importance depends on the condition of the patient) and the theory behind ECG interpretation is vast, making this an interesting field of algorithm theory. The main reason for not interpreting more signals is simply

the magnitude of the task and that this is not necessary for a proof-of-concept use.

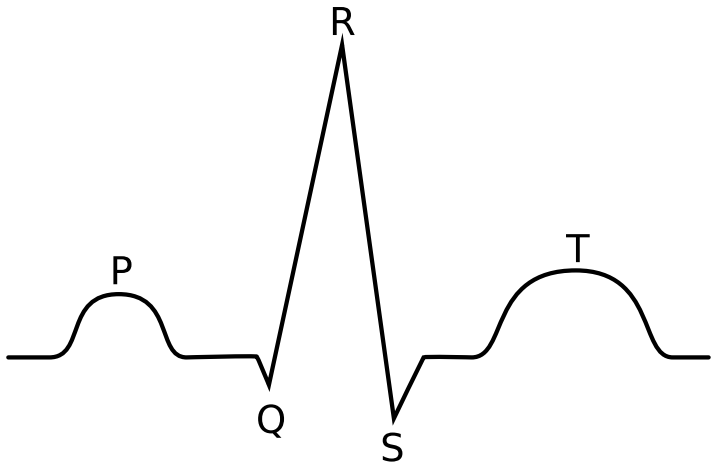


Figure 6.1: ECG model.

ECG signals are output from an *electrocardiograph*, which measures electrical voltage in the heart[15]. For a cardiologist, ECG signals provide much information about a patient's heart condition, making ECG interpretation an important field in modern medicine. As ECG signals are continuous data plots, there is great need for automation of the monitoring process. This represents a whole field in algorithm theory; pattern recognition and detection of peaks in ECG signals.

ECG graphs have a distinct, characteristic shape, which can be seen in Figure 6.1. The reference points P, Q, R, S, and T, represent the following electrical signatures from a heartbeat (see figure 6.2 for an illustration of the terms):

- The **P** wave is produced by the current that causes contraction in the left and right atria.
- The **QRS** complex is made by contraction of the left and right ventricles. As these contractions involve more muscle mass and are more forceful than the P waves, they produce a much larger fluctuation in the graph.
- The **T** wave is caused by *re-polarisation* of the ventricles, a process analogous to resetting a spring after it has been released.

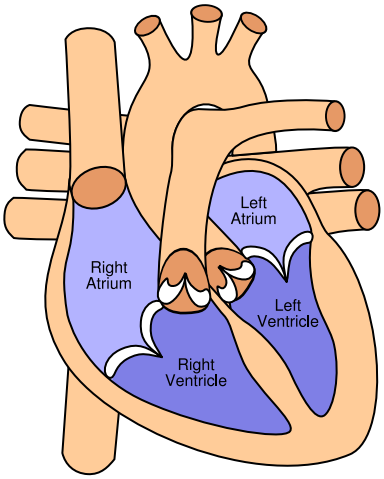


Figure 6.2: The human heart[18].

For the cardiologist, the shape of the ECG signal is very informative, but automating the interpretation process is difficult. The shape of ECG signals vary tremendously (Figure 6.3 has some examples) and can sometimes take on forms almost completely unrecognisable to an untrained eye. Some patterns are easier to locate, though, the QRS complex probably being the most important, enabling medical equipment to establish a patient's heart rate by itself. An example of another important pattern to scan for is the distance between the points Q and T, namely the QT interval. A prolonged or shortened QT interval can be an indication of the hereditary conditions *Long QT Syndrome* or *Short QT Syndrome*[5].

6.1 QRS detection

The largest, regular, fluctuations in ECG signals are the QRS complexes. They represent the contractions of the ventricles — the largest muscles in the heart — and are used for measuring the heartbeat frequency, or pulse. While other elements of ECG signals are also important and can provide cardiologists with vital information, they are not as easily seen.

Recognising shapes in data is often a difficult computational task, requiring advanced algorithms, and when the data is noisy and depicts natural phenomena (as opposed to being the result of a

mathematical function) it is even more difficult. Natural data is unpredictable and as ECG data is measurements of electricity, with relatively small variations in voltage, noise can be significant.

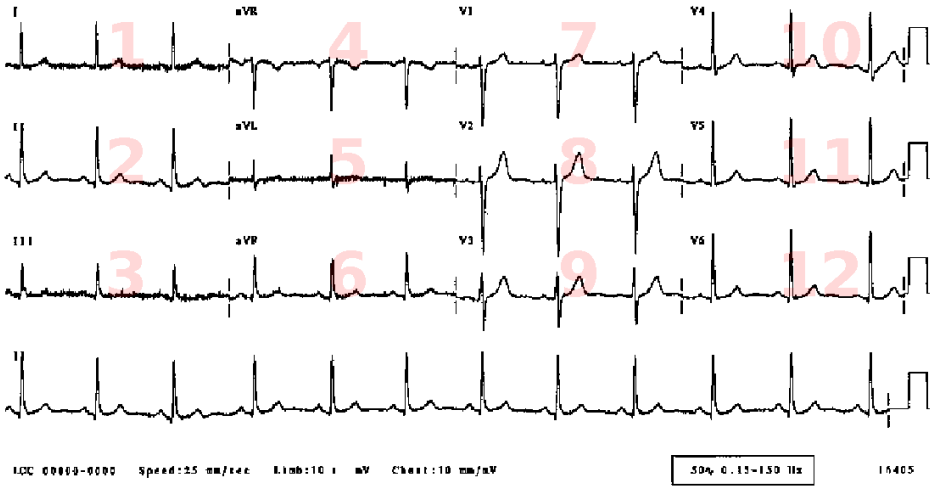


Figure 6.3: 12 lead ECG (figure from [23]).

In order to better be able to filter out what one needs, it is customary to use multiple leads when measuring ECG. A 12 lead ECG (see Figure 6.3) is a common means of retrieving detailed information about patients' hearts' conditions. Several different electrodes positioned around the body, standard 12-lead ECG with one on each arm and leg and six on the chest (see Figure 6.4), are combined into 12 leads that provide a very detailed picture of a heart[4][1]. For monitoring purposes, fewer electrodes and leads are often used, but to obtain a good picture, more than one is necessary.

6.1.1 Modern algorithms

Among the many algorithms for pattern recognition in ECG signals, combining the leads is a common way to reduce the effects of noise and get a cleaner and clearer picture of the actual voltage fluctuations caused by the heart. Filtering the data, for example

[1]The 12 leads are generated by 10 electrodes or less. This as the leads are produced by measuring the current between two electrodes and the combination of such pairs decide the number of leads.

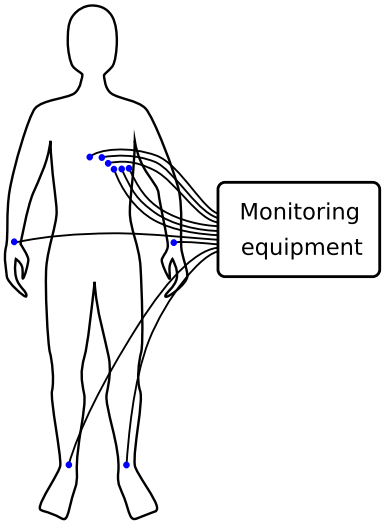


Figure 6.4: Approximate placement of electrodes for standard 12-lead ECG.

using noise reduction filters, and approximating differentiators helps smooth the resulting graphs. Furthermore, calculating absolute values from the measurements is a means of producing graphs in which it is easier for an algorithm to "see" the QRS patterns.

An important factor in QRS detection is whether the process needs to be done in real time. Study of recorded data is both common and important, so all algorithms do not necessarily need to run in real-time and detect QRS complexes as they occur. When data is recorded beforehand, an algorithm can be allowed to look at larger portions of the graph before making a decision as to where the QRS complexes occur. One can look at the whole data set or just a portion of it, but, most importantly, one can look "into the future", at measurements that were made after the current. In this way, it is of course easier to distinguish a slope in the graph caused by other factors than for instance a QR slope as one can see how the graph develops further on. Real-time algorithms do not have this luxury. With recorded data, running time for an algorithm is also not very important and one can use detection algorithms with arbitrary complexity (of course, one still has to be able to get a result while it is still relevant). Again, real-time algorithms do not share this luxury and are bound by the computational power of a given computer and algorithm.

For clinical applications, modern real-time QRS detection al-

gorithms can provide a detection accuracy of about 99.5% without too much computational effort, which is satisfactory for clinical applications. For research and study purposes, a higher accuracy may be desirable, and this is provided by more complex and time and computationally intensive algorithms.

6.1.2 Real time electrocardiogram QRS detection using combined adaptive threshold

With an abundance of algorithms to choose from, the following criteria laid the basis for the selection, although for a proof-of-concept implementation, some of these could be disregarded in part or in whole:

- High detection accuracy.
- The algorithm must be real-time, this is an absolute demand.
- Sampling frequency independence is desirable.
- Relative ease of implementation.

A modern algorithm, published in [2], meeting these criteria is *Real time electrocardiogram QRS detection using combined adaptive threshold*. For these reasons, and because it is a modern algorithm, it was chosen as the QRS detection algorithm to implement.

First, the data undergoes three preprocessing steps, the first two are filters to reduce noise and smooth the data. The exact algorithms used in these filters are not described, but a type of smoothing filter is further explored in Section 6.1.3. Then, the L different leads are combined into a *complex lead* using Equation (6.1), which is calculated for each measurement i.

$$Y(i) = \frac{1}{L} \sum_{j=1}^{L} |X_j(i+1) - X_j(i-1)| \tag{6.1}$$

The complex lead is a combined absolute value of an approximation of the differentiated leads providing a graph in which variations are amplified and QRS shapes are more easily seen.

After one has obtained a complex lead, it is easier to find the QRS complexes, but it can still be difficult to sort out small complexes, and if the number of leads is small (the algorithm can work with only one lead), noise can still cause problems. To further improve detection, the complex lead is used to calculate an adaptive threshold that follows the ECG graph and when the ECG signal rises above the threshold, a QRS complex is recorded. The threshold is calculated as a sum of three calculations:

Adaptive steep-slope threshold

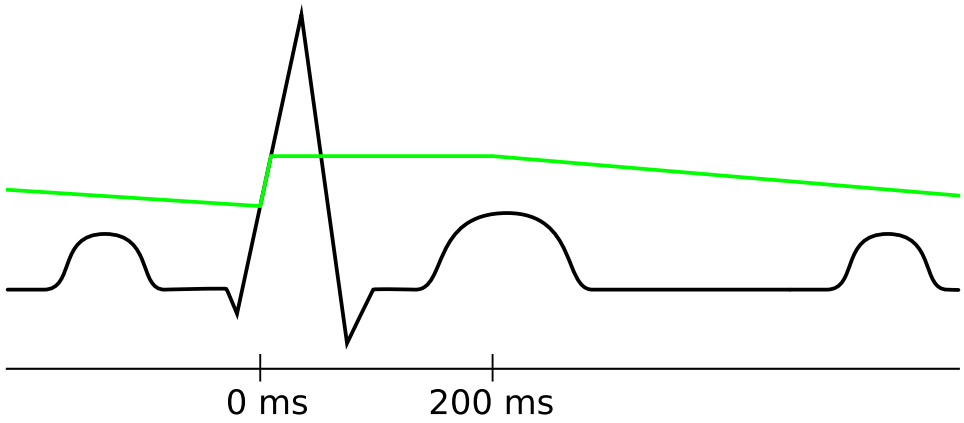


Figure 6.5: Adaptive steep-slope threshold.

Denoted M, the *adaptive steep-slope threshold* is an average over a buffer of five values calculated from previously recorded steep slopes (the steep slopes are the QR part of the QRS complexes). After a steep slope is detected, M remains the same for 200 ms. After this, M gradually decreases in value until 1200 ms after the QRS detection, where it is reduced to 60% of it's value at the time of detection (see Figure 6.5). No detection is allowed for the first 200 ms after the previous detection and if a detection occurs before 1200 ms after the previous, M stops its decline and is refreshed.

Adaptive integrating threshold

Designed to reduce the effects of electromyogram noise — noise induced by the measuring equipment, *the adaptive integrating threshold*, denoted F, is calculated using values from the preceding 350 ms of the current signal. The maximum value from the first

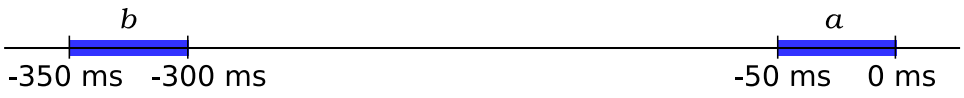


Figure 6.6: Adaptive integrating threshold.

50 ms in this interval is subtracted from the maximum value from the last 50 ms. This is illustrated graphically in Figure 6.6, where the intervals are labelled a and b, and stated formally in Equation (6.2).

$$F = F + (\max(Y_a) - \max(Y_b))/150 \tag{6.2}$$

Adaptive beat expectation threshold

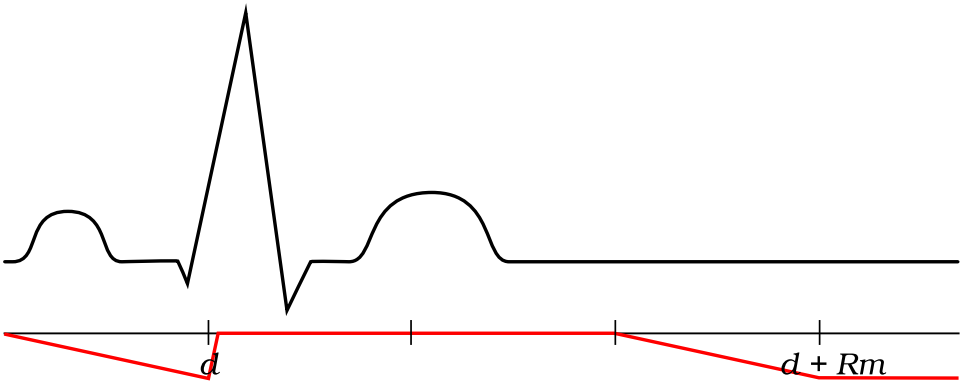


Figure 6.7: Adaptive beat expectation threshold.

To help detect smaller amplitude beats following normal or large amplitude beats, the *adaptive beat expectation threshold*, denoted R uses the average RR time, RR being the time from one R peak (in a QRS complex) to the next. A buffer with the last 5 RR distances is kept and the average value, Rm is used when calculating R.

After a QRS complex is detected, R remains at 0 mV for the duration of $\frac{2}{3}Rm$, after which it decreases 1.4 times slower than M decreases in the interval from 200-1200 ms after a QRS detection. After another $\frac{1}{3}Rm$, R's decline is stopped. The R threshold is illustrated in Figure 6.7.

Combined adaptive threshold

A result of the three thresholds, M, F and R, the combined adaptive threshold is calculated as $MFR = M + F + R$. As previously mentioned, when the ECG signal rises above MFR — when $Y(i) \geq MFR$, a QRS complex is detected.

6.1.3 Implementation

Preprocessing filters

As mentioned in Section 6.1.2, The first two preprocessing steps in [2] were not described, but the purpose of these is to smooth the data and reduce interference. I implemented a *FIR* filter in a test application, but due to time constraints, this was not incorporated into the QRS detection algorithm in the monitoring application. This filter is defined by Equation (6.3) and illustrated in Figure 6.8, where $m + 1$ measurements are multiplied with $m + 1$ coefficients and combined into a filtered measurement. After each filtered output, the measurements are shifted, another is added, and a new filtered output is calculated.

$$y_n = \sum_{k=0}^{m} c_k x_{n-k} \tag{6.3}$$

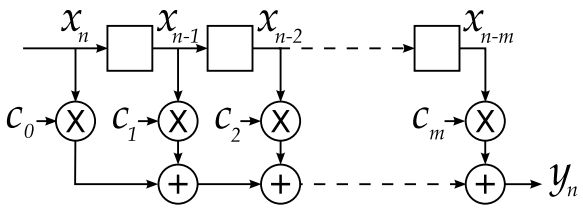
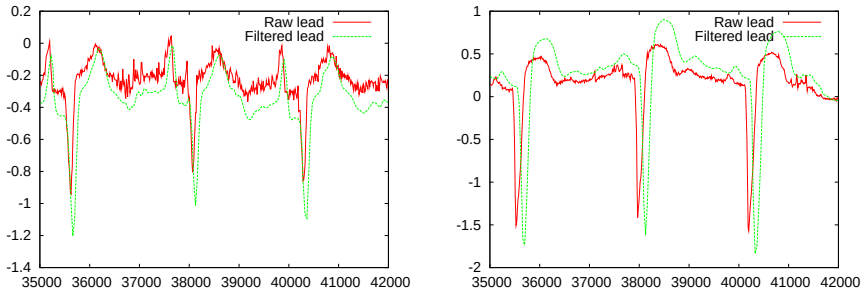


Figure 6.8: FIR smoothing filter.

Due to the aforementioned time constraints, the coefficients and length of the filter were not refined, but further testing will improve in these. Some tests of the algorithm was performed, however, and an example with $m = 9$ and using the coefficients $\{0.05, 0.1, 0.15, 0.2, 0.25, 0.25, 0.2, 0.15, 0.1, 0.05\}$, is given in Figure 6.9.



(a) Lead I.

(b) Lead II.

Figure 6.9: Example of using a smoothing filter to reduce interference.

QRS detection algorithm

Apart from the preprocessing filters, the algorithm described in Section 6.1 was implemented without difficulties, with one exception: In [2], the peaks of the QRS complexes in the complex lead are plotted as single, clean peaks. In my implementation, these peaks are cut in half, with a decline instead of a peak, as can be seen in Figure 6.10.

The way the complex lead is calculated, this interruption in the peaks is not unexpected, as Equation (6.1) calculates the difference between the points after and before the current point. If the current point is at a symmetric peak, then the difference between these points will be 0, resulting in a sharp decline in the graph. The reason for the difference between my implementation and the published results is unclear, but it may be caused by the preprocessing filters used. This does not keep the algorithm from working, but it is likely that it affects its performance.

In the case of further development of the monitoring application, this deviation should be investigated further if the same QRS detection algorithm is used.

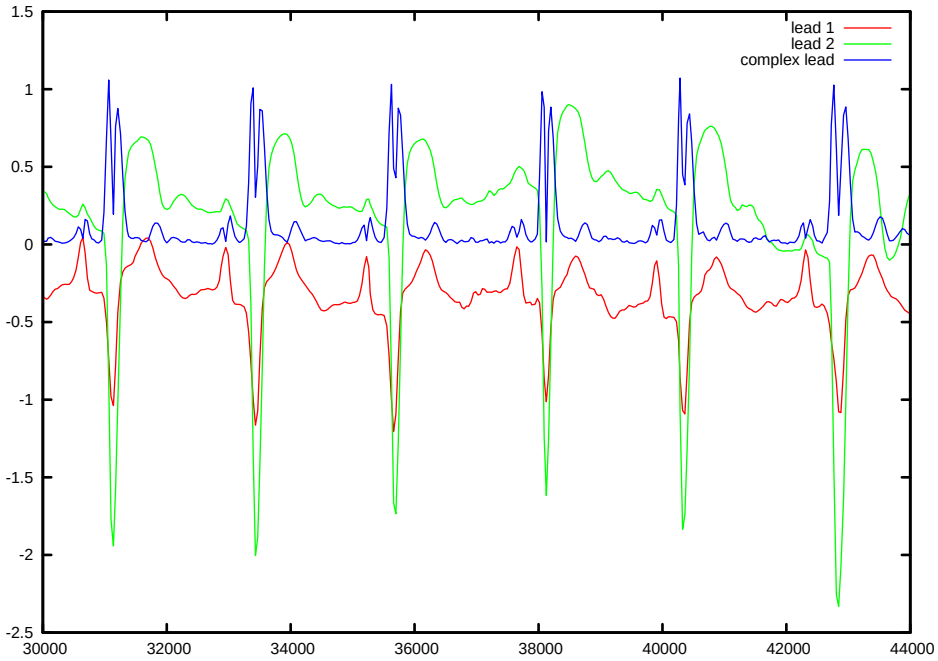


Figure 6.10: Calculated complex lead.

Chapter 7

Software design

In general, software design is a vast area with many paradigms living side by side, some complementing, and some contradictory to, others. While all such paradigms have their followers and critics, two important goals of many paradigms are:

- *Reusability*. When code is written, much effort is put into making it right with regard to robustness, speed, and other factors defining "high quality" code. Since this takes a lot of time, one does not want to produce more code than necessary and making code that can be reused or reusing already written code is good practise.
- *Modularity*. Instead of making "monolithic" code with lots of functionality written into it, it is better to make smaller modules with well defined functionality and suitable interfaces, and put these modules together in order to create a whole. Modularity is a good thing for two reasons; firstly that it is easier to reuse smaller modules than monolithic code, and secondly that this leads to better robustness and verifiability. It is much easier to see that smaller modules do the job they are supposed to do than figure out which part of a monolithic system does what and thus easier to debug and also of course, reuse.

While for this project, a specific QRS detection algorithm, described in Chapter 6, was used, other algorithms may prove to be better for specific uses, or of course also in general. Separating the QRS detection algorithm into a separate module therefore adheres to

the design principles mentioned and provides code that is easier to rewrite and easier to debug. The part of the code that creates the plots and makes the necessary calculations should be separated into library modules. Since this part of the code is in an early phase of development it can be fruitful to try different models, something a rigid division could complicate. With regard to future work and further development, however, this should be taken into account.

More specifically, another aspect of software design to take into account when programming for the PDA environment is how to divide the work between modules and devices with regard to the work load of specific tasks and efficiency in general. While modularity can provide reusability and robustness, it can cause a decline in efficiency as generality often comes at the price of incorporating functionality not needed by everybody. This can be perfectly acceptable on most workstations, but become critical when the platform has limited memory and computational resources.

In Figure 7.1, the monitoring application can be seen in its current state. It is a screenshot from a PDA where four different leads provide four plots, the two latter are ECG signals.

This chapter will focus on how the application handles the data *representation*, as opposed to data transportation and interpretation from earlier chapters. We start with how data is plotted and then go into detail about how the data is transformed from their raw measurements to plottable coordinates. In this context, I will refer to *models* and *scenarios* where a scenario is a particular setup of devices, that is, a client and server or just a client. A model denotes a specified division of tasks between the devices involved in a scenario.

7.1 Processing and displaying data

In all models, each data point measured is mapped to its corresponding coordinates in the plot, the function (y) value is calculated using the height of the plotting area and the preset maximal and minimal values a data set can have. The time (x) value is set using a preset *data resolution*, where a certain amount of data is plotted within a given time span. Because the screen of a PDA is small compared to monitors, compressing the plots is necessary

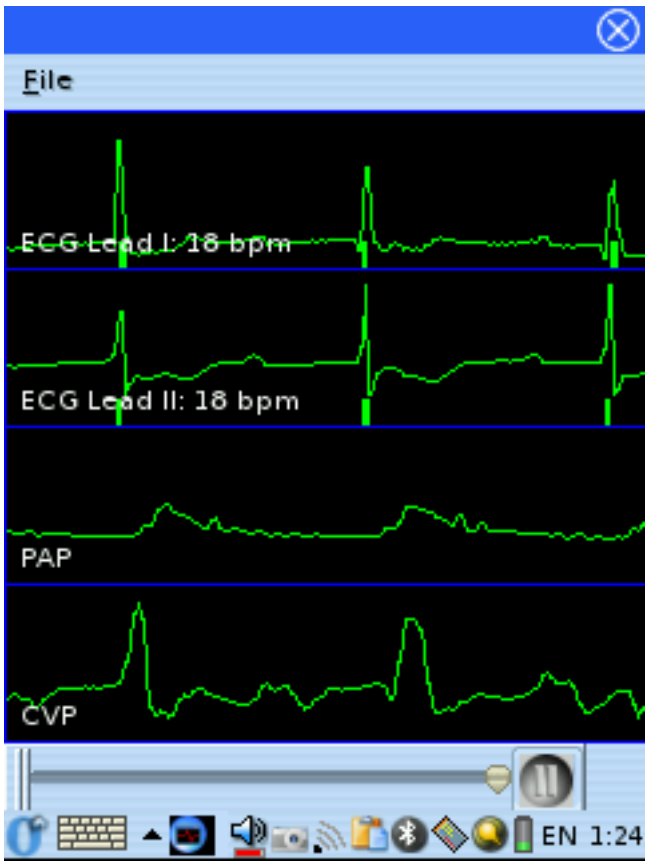


Figure 7.1: Screenshot of monitoring application, currently showing two ECG leads, *pulmonary arterial pressure* (PAP) and *central venous pressure* (CVP).

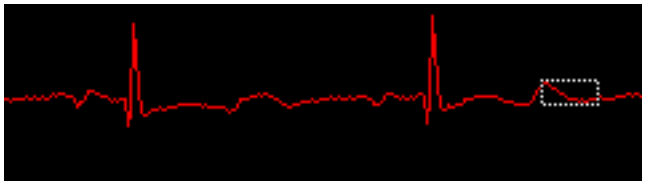


Figure 7.2: A plot from the monitoring application (dotted region expanded below).

to fit as much of the signal into the screen at a time as possible. Therefore, each time unit is plotted on one pixel. Figure 7.2 shows part of a plot of an ECG signal, the dotted region is expanded in Figure 7.3, where each pixel is visible. The latter figure is divided into one-pixel width segments for the sake of the example.

In Figure 7.4, it is shown how the plot is updated. The leftmost

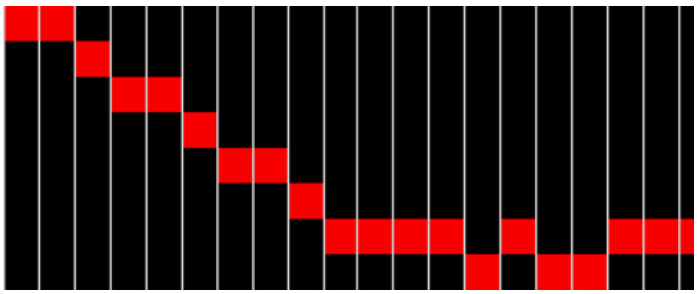


Figure 7.3: A closer look at a plot.

pixel (to the left of the blue line) is removed and the rightmost pixel (to the right of the green line) is added. In this way, the plot is always refreshed and it is simple to maintain, using as little memory as possible while still keeping the painting requirements to a minimum.

Figure 7.4: Updating the plot: The rightmost pixel is added and the leftmost removed.

Automatic interpretation of measurement data is, as stated in Chapter 6, only implemented for ECG signals, as pattern recognition is a difficult task and implementing algorithms is time consuming, leaving interpretation of other types of signals outside the scope of this thesis. Before the coordinates of measurements are calculated, ECG measurements can be input to the implemented QRS detection algorithm which provides the heart rate. This again can be used for setting off alarms if the frequency is outside a given range.

In this context, the client is the PDA and the server is a stationary computer able to process much more data than a PDA, communicating with the client over some form of network connection, in the case of the monitoring application, preferably a wireless connection using the tcp/ip protocol.

There are several possible ways to design such an application, with the client doing different amounts of the computing necessary. The simplest models do not use a dedicated server and the client continuously reads raw data, provided by the measuring instruments or possibly some intermediate device, from sockets. Thereafter, the client does the necessary calculations and updates its plots. Here, there are two further models that have been studied and tested, namely the client performing calculations of the plotted point for each data point read and a model where the client performs all necessary calculations once, and then uses these results when later plotting data.

Another scenario is with a dedicated server performing all necessary calculations and formatting the data in a way that is efficiently read and presented by the client. This relieves the client's lesser memory and computational power from a lot of work that is much easier done by the server. A more detailed description of the three models follows below.

7.2 Client only scenario

In this scenario, there is no server, and the client receives data on different ports from the instruments monitoring the patient, either directly, or via some device that simply serves as a bridge, writing the data to a specified port. This scenario is exemplified in Figure 7.5.

7.2.1 Computationally intensive model

Every measurement is compared to the height of the plot area and the preset range that encloses the measurements and it's corresponding coordinates are calculated. Furthermore, for ECG signals, the measurements can be input to the QRS detection algorithm to obtain the heart rate.

The advantage of this model is that it saves memory. There is little need to save data to memory and one can therefore theoretically combine many different measurements. The disadvantage is that all calculations are heavy for such a relatively weak CPU and the amount of possible simultaneous plots is strictly limited.

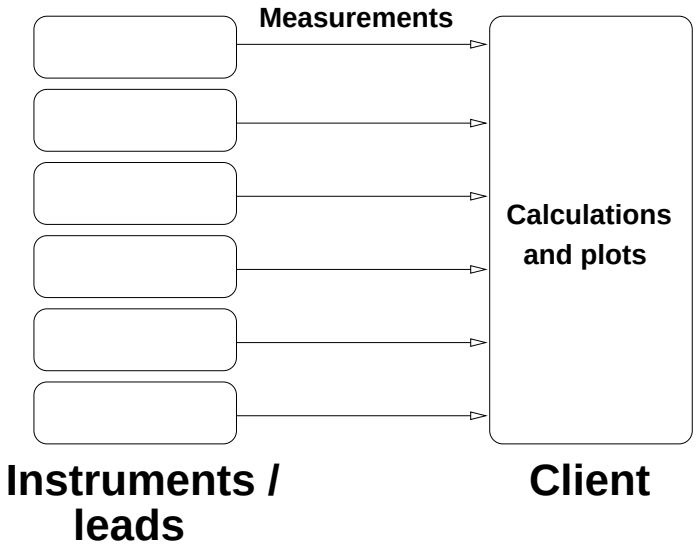


Figure 7.5: Client receives, processes and presents raw data.

Furthermore, as is explored further in Section 8.1.2, the ARM CPU has no dedicated floating point processing hardware. This forces floating point calculations to be done in software, making the use of floating point numbers expensive.

This implementation proved too costly, after adding three signals, the quality of each signal deteriorated notably and after adding a fourth, only unintelligible remains of graphs remained. Requiring the input to be on integral form to ease floating point calculations did not solve this problem.

7.2.2 Computationally simple model

In an attempt to reduce the computational load for the PDA and further reduce calculation overhead, another model was implemented. This model requires the measurements to be on discrete intervals giving unique mappings *measurement* → *data point*[1]. One starts with the preset range enclosing the maximal and minimal allowed values for the signal and the height of the

[1] One could also imagine a model where the measurements were mapped to coordinates in ranges, i.e. measurements $[i, j\rangle \rightarrow y$ coordinate k etc., but as the discrete interval model did not improve much on the results, testing this proved unneccessary

plot area. From this, all possible coordinates for all possible measurements are calculated and put in a mapping data structure with the measurements as keys and coordinates as values. As long as the number of possible measurements is relatively small, this relieves the client of having to do apply these calculations for all measurements.

With this model, the number of necessary computations is strongly reduced. However, all possible mappings *measurement → data point* must be stored in a map data structure, making the use of large precision floating point numbers difficult and necessitating the storage of the calculated maps.

This model did not improve noticeably on the previous, although it was tested even with small ranges (only 50 possible measurements), producing small mapping structures, so this solution is also flawed. The conclusion is that this type of work is too costly for the type of PDA used and another design is needed.

7.3 Client — server scenario

As the processing and memory capabilities of a PDA are fairly limited, a model in which the client does as little work as possible and leaves as much as possible to a dedicated server may prove itself to be advantageous. Both the previous models put a large workload on the client; the client only receives the measurements, and must itself translate these measurements to points and make all necessary calculations based on these.

A workstation has much more calculation capabilities than a PDA, since it is possible to equip it with faster and more CPUs, more RAM etc. It is therefore much easier for the workstation to perform calculations and when calculations produce a bottleneck, the workstation is better at solving the problem.

As can be seen in Figure 7.6 on the following page, in this model, the server collects all the raw data from the different instruments. When the client connects to the server, it reports its available height to the server, which divides this height among the active data sets. Then the server starts translating the measurements into coordinates, which it sends to the client, relieving the client of much of its previous workload.

The server makes all necessary calculations of coordinates, resampling the data if necessary and it also handles QRS detection and, if implemented, other data interpretation functions.

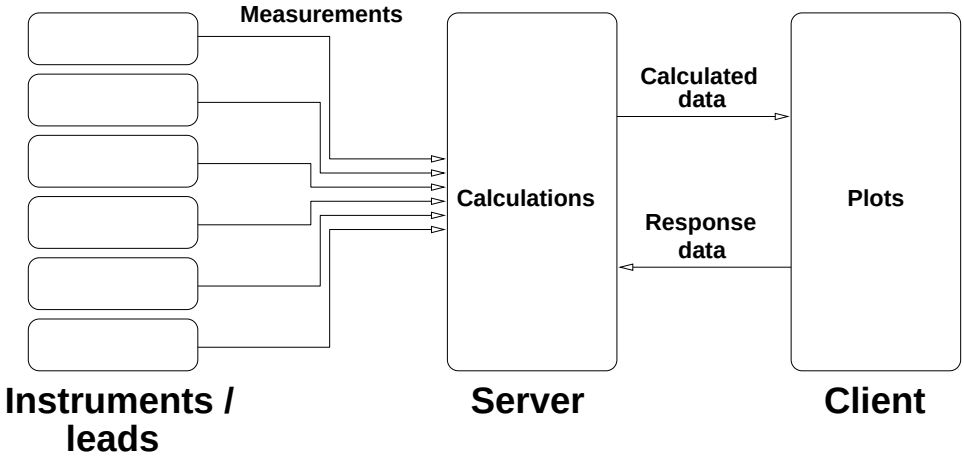


Figure 7.6: Dedicated server performing most calculations.

Chapter 8

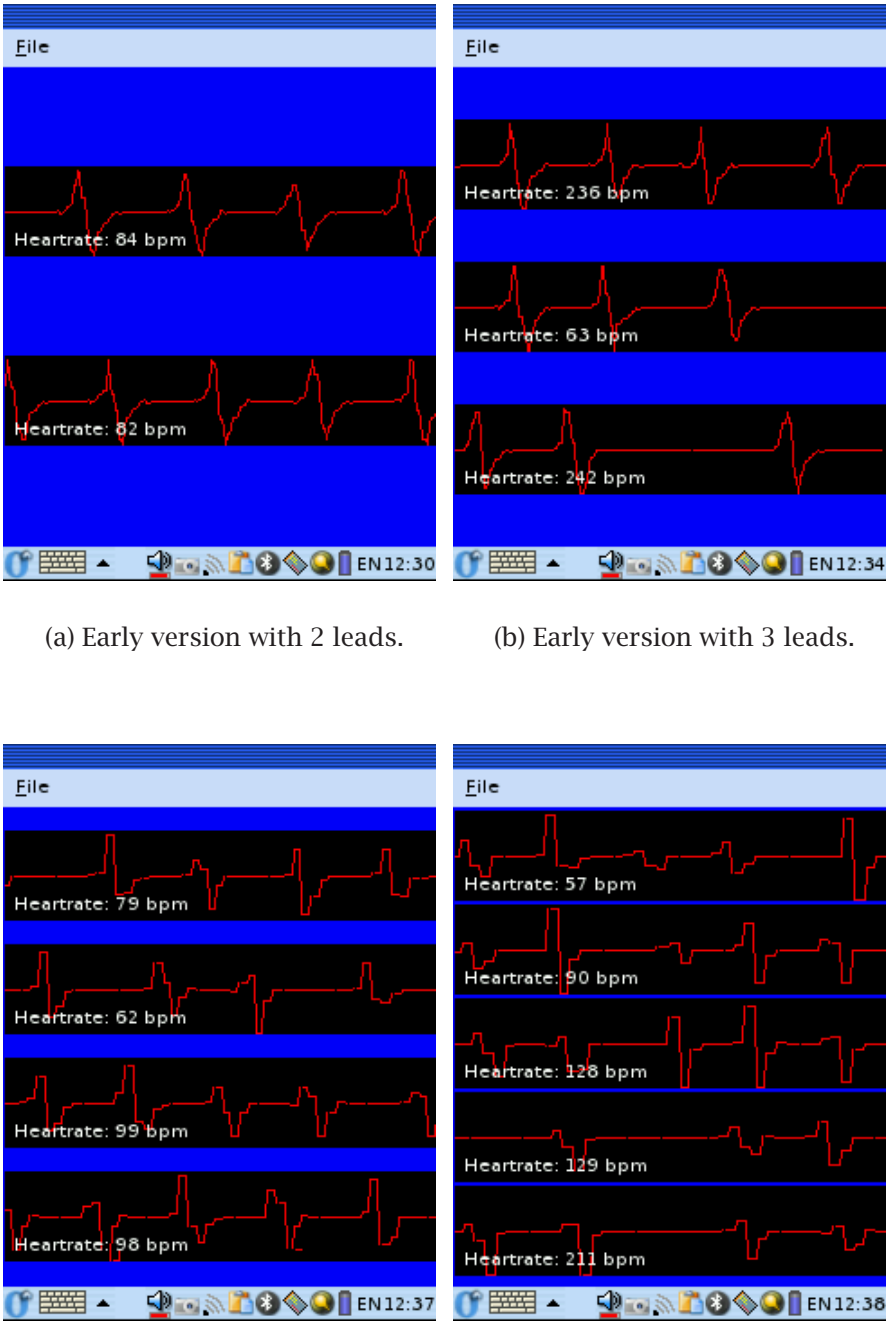
Tests and results

The need to make a transition from a client-only scenario into a client-server scenario (see Section 7.1) was originally made apparent by trial (and error). The idea was originally to construct a more or less self-reliant client application for use on portable devices, but the need for more computational power than the iPAQ could provide at the time of development necessitated such a division of work between the client, which now primarily serves as a user interface and display for the whole system.

At an early state of development, I had no real life test data available, so the data used to create the plots was a simple, recurring graph, generated by a shell script. This script was later rewritten as a C++ program in order to write the data to multiple sockets instead of either to the *standard out* or the *standard error* output stream[1] as two output streams was not sufficient, and sockets are the means by which communication is normally carried out on Unix systems.

In Figure 8.1 can be seen four screenshots of the monitoring application, with an increasing number of leads displayed. The data plotted in these screenshots is from the test application and not real-life data. In Figure 8.1(a), only two leads are presented, and they are displayed fine (provided the PDA has a low work load from other processes). With three leads (Figure 8.1(b)), the data is still relatively smooth (the leads are independent of each other, so the fact that the peaks are at different places does not matter). Only

[1] On Unix platforms and derivatives, both standard output and standard error are output streams normally going to the terminal, but can be redirected to files and other devices. They are usually present for all programs to use.



(a) Early version with 2 leads.

(b) Early version with 3 leads.

(c) Early version with 4 leads.

(d) Early version with 5 leads.

Figure 8.1: Early version of monitoring application in a client-only scenario. The deterioration of the plots is clearly visible as the number of leads increase.

when another lead is added (Figure 8.1(c)), can we see that the plots become very distorted and by adding yet another (Figure 8.1(d)), the plots become almost unintelligible.

At this point it is worth noting that the application at this stage was stripped of floating point calculations as much as possible and thus only accepted integral input. Also, the QRS detection was merely done by looking at a flat threshold: whenever a peak occurred above this, a beat was detected. This is a very simple form of peak detection which is totally inadequate with real ECG data, with which more sophisticated QRS detection algorithms must be used. These, more sophisticated algorithms, requireme more computational power from the system and makes detection even more costly, further degrading the performance of the application.

8.1 Benchmark tests

In order to further investigate the capabilities of the iPAQ, several benchmark tests were performed. When using benchmark tests, one normally wants to compare the running system to others, but many factors influence the results. For one thing, when benchmarking a system running Linux, the kernel is important. Newer kernel versions frequently bring improvements to system performance and affects benchmarking results.

Another important factor is what one wants to measure. A computer comprises many components, and one often wants to compare a specific part of the system, such as memory or computational capabilities. The problem is that components affect each other's performance, so in the case of memory tests, a faster CPU and data bus will also improve the results.

Benchmarking is generally divided into two different families of tests: *synthetic* and *application* benchmarks[1]. Synthetic benchmarks are designed to test a specific component, like the performance of RAM memory or the CPU, but such one-sided usage of a computer is rarely seen in real applications. This kind of tests can give an impression that is a skewed image of the performance of a system. Application tests measure overall performance and give a better impression of the system's total capabilities. Timing the compilation of a Linux kernel is an example of an application benchmark that shows the capabilities

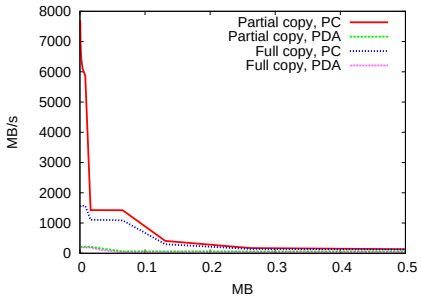
of a system as it incorporates most procedures found in real-life applications. Generally, both types of tests complement each other and to obtain a good picture of a system's performance, the should be used in conjunction.

That said, we already know that proper workstations perform better than PDAs and the purpose of these tests is not to establish the exact scale of the difference, but to obtain a clearer picture of what is hardest for the PDA to do and an idea of how much can be expected from it. The comparison is not quite fair, a Pentium III 736.16MHz CPU, 896MB RAM workstation was used to compare with the iPAQ. This in itself gives skewed results, but to some extents does show some of the differences between a workstation and a PDA. The tests performed are more of a synthetic type than an application type, for two reasons: For one thing, the monitoring application works as an application benchmark, as it incorporates many common types of operations, like computations, memory usage, data transfer and drawing to the screen. Secondly, a full comparison of the systems is not that interesting, we want to have a look at which operations are most expensive and for this purpose, synthetic benchmarks provide specific, but good answers.

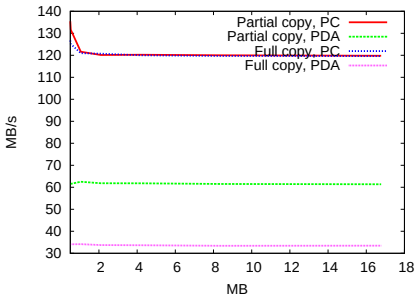
All tests were performed three times with the presented results being the average. No other processes ran in the foreground, but on the workstation (PC), an X server ran and on the PDA, OPIE ran.

8.1.1 Benchmarking memory

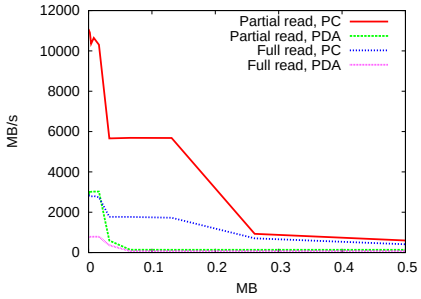
lmbench is a suite of benchmarking tools (http://www.bitmover.com/lmbench/) that, among other things, can test CPU and memory capabilities. The results of the memory bandwidth — how much memory can be moved — tests can be seen in Figure 8.1. The interesting specifications of the two systems compared, denoted *PC* and *PDA*, are listed in table 8.1.



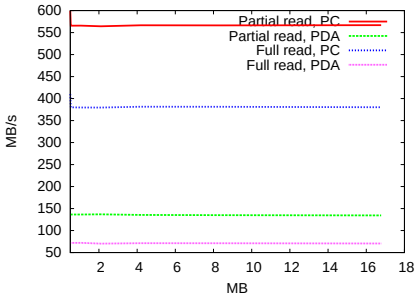
(a) Copying data from one location of memory to another, 0 — 0.5MB.



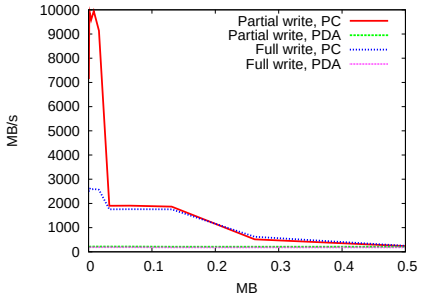
(b) Copying data from one location of memory to another, 0.5 — 16.78MB.



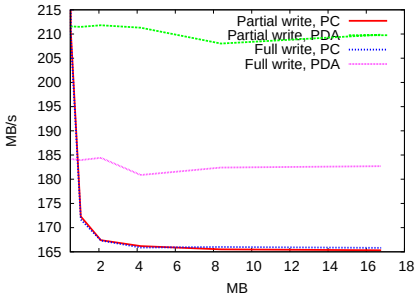
(c) Reading data from memory, 0 — 0.5MB.



(d) Reading data from memory, 0.5 — 16.78MB.



(e) Writing data to memory, 0 — 0.5MB.



(f) Writing data to memory, 0.5 — 16.78MB.

Figure 8.2: Results from memory testing. (System specifications can be found in Table 8.1.)

	PC	PDA
CPU	Pentium III 736.16MHz	XScale 398MHz
L1 data cache	16kB	32kB
L1 instruction cache	16kB	32kB
L2 cache	256kB	-
RAM	896MB	64MB
Linux kernel	2.6.12	2.4.19
Linux distribution	Debian	Familiar

Table 8.1: PC compared to PDA.

Figures 8.2(a), 8.2(c) and 8.2(e) contain the measurements from tests running from 512B to 0.5MB of data through several tests. Figures 8.2(b), 8.2(d) and 8.2(f) are the same data, only from 0.5MB to 16.78MB, in separate figures to improve visibility. A simplified view of the test code explains more about what is going on.

The figures 8.2(a) and 8.2(b) display the results of copying tests, where data is copied from one location of memory to another. Simplified code for the partial version looks like

```
for (i = 0; i < N; i += 4)
  dest[i] = source[i];
```

and the full version is

```
for (i = 0; i < N; i++) {
  sum += array[i];
  array[i] = 1;
}
```

Figures 8.2(c) and 8.2(d) are from read tests. Data is read and summed up from a portion of memory. The partial version's simplified code is

```
for (i = 0; i < N; i += 4)
  sum += array[i];
```

The full read version is the same, except i is increased by 1 instead of 4.

Writing to memory (shown in figures 8.2(e) and 8.2(f)) is tested by (partial version)

```
for (i = 0; i < N; i += 4)
  array[i] = 1;
```

and similarly, in the full version, i is increased by 1 instead of 4. The significant peak of throughput on the PC, with small amounts of data (512B - 16kB) is probably caused by the *L1 cache*[2] as this cache is large enough to accommodate the entire memory section. The same peak is not as notable on the PDA, presumably caused by the L1 cache of the Pentium being faster than that of the XScale. The peak on the PDA is, however, slightly longer, as the L1 cache of the XScale CPU is 32kB, twice the size of the Pentium one, after which a drop can also be seen here.

The XScale lacks an L2 cache so the results level out over 32kB. Here, the Pentium performs better as it has an L2 cache of 256kB, enabling quicker memory access than the XScale, which at this point needs to read and write to RAM.

Over 256kB, we see that the PC's memory operations generally are quicker than those of the PDA. Reading and copying in memory is approximately 2 — 5 times quicker on the PC. Since the Pentium has almost double the clock frequency of the XScale, much of this is to be expected, and the results are more or less as expected.

When testing writing bandwidth, however, the XScale suddenly outperforms the PC when the cache is depleted, particularly when only writing to every fourth array index. This could be caused by different *write policies*[3] on the caches or other CPU specifics. I was not able to investigate this further, but it is interesting and should possibly be explored further.

8.1.2 Benchmarking CPU

Since the XScale does not have floating point processing hardware, the testing of the CPU is divided into two parts, one testing with, and the other without, floating point calculations.

[2] In the Pentium III CPU family, there are two levels of cache, 1 and 2 (L1 and L2). L1 is significantly smaller than L2, but is quicker.

[3] A CPU writes to memory via the cache, using a *write policy* that decides whether data is written directly to RAM after being written to the cache (*write-through*), only after the cache has been depleted (*write-back*) or something in between.

Testing floating point performance

As the XScale CPU lacks dedicated floating point hardware, its capabilities with regard to computations involving floating point numbers is greatly reduced compared with most workstations. Most modern general purpose CPUs have FPUs (Floating Point Units) and are thus much better suited for such operations.

For investigating the floating point capabilities of the XScale CPU, a short C program was written, calculating prime numbers. The algorithm involves some real-life computing, including floating point operations. It is not intended to be efficient and is basically written as follows, with p being the prime numbers, finding the first N primes:

```
p = 2;

for (i = 0; i < N; ++i) {

    if (p == 2)
        a = p + 1;
    else
        a = p + 2;
    b = 3;
    roof = sqrt ((double) a);

    do {
        if (a % 2 == 0 ||    /* a is an even number */
            (a != 3 &&       /* a is not 3 and */
             a % b == 0)) { /* is divisible by b */
            b = 3;
            roof = sqrt ((double) ++a);
        }
        else
            ++b;
    } while (b <= roof);

    p = a;
}
```

As expected, and as can be seen in Figure 8.3, this algorithm performs significantly better on the Pentium, while the XScale is struggling with only approximately 1/100 of the performance.

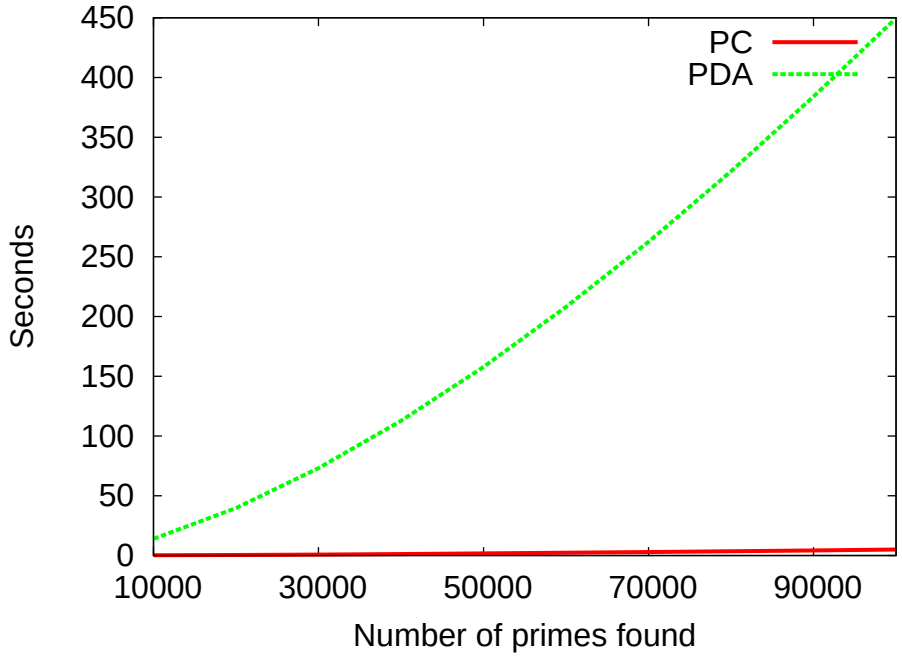


Figure 8.3: Results from CPU testing using a prime number generating algorithm.

A short program calculating $\sin x$ is outlined as follows:

```
sign = -1

for (i = 3; i < 2 * N; i += 2) {
    res += sign * (pow (x, i)/ fac(i));
    sign *= -1;
}
```

with `pow` being the exponential function and `fac` being the factorial function. This test does also use floating point calculations and the results are given in Figure 8.4.

Testing non-floating point performance

To separate what part of the results of the floating point tests is caused by the lack of dedicated floating point hardware, the prime number generating program was slightly rewritten so that it doesn't use the FPU. This rewrite involved removing the calculation if the square root of a and allowing b to rise up to a instead if a's square root. Of course, this is highly inefficient, but that is a good

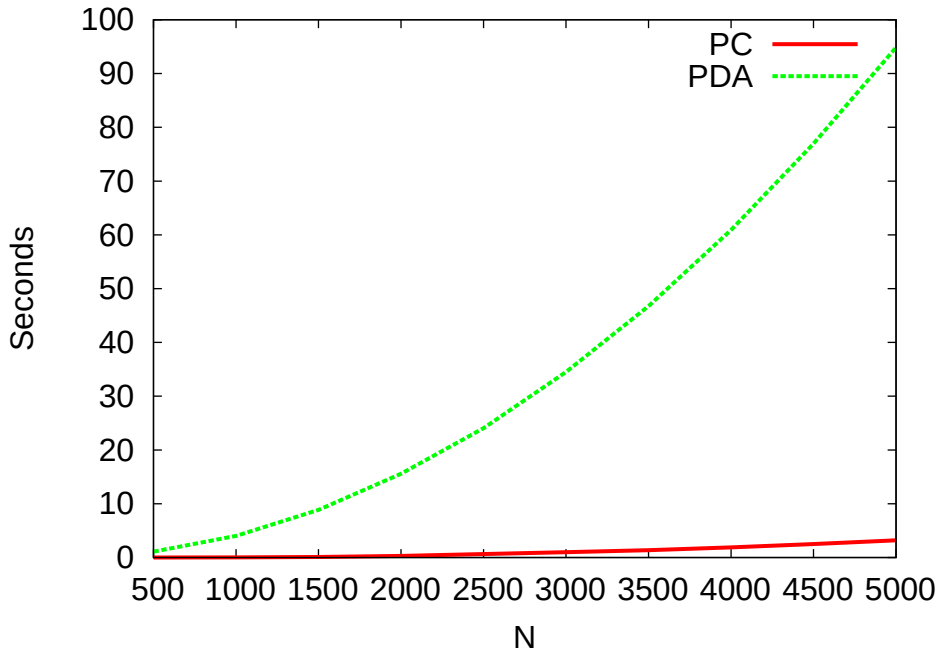


Figure 8.4: Calculating sin(1).

thing when running performance tests like this as it gives the CPU much work.

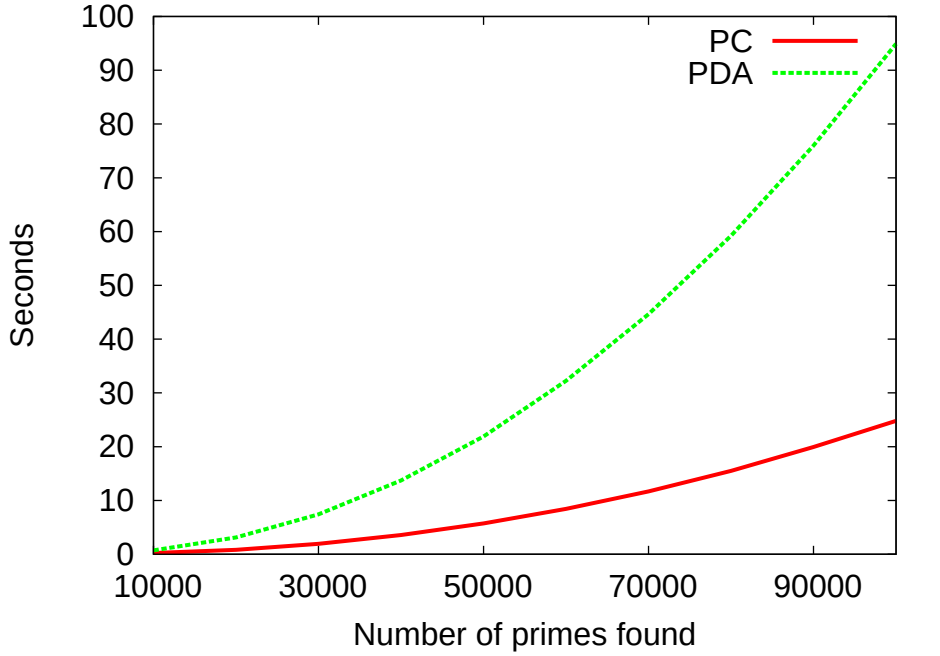


Figure 8.5: Calculating prime numbers not using the FPU.

As can be seen in Figure 8.5, the difference in performance between the different architectures is much smaller than with programs using the FPU. This indicates that the lack of an FPU can be a problem in programs with high requirements to efficiency and real-time performance, as long as floating point operations are required. Indeed, the performance of the XScale is over 4 times better when omitting the square root calculations, even though this means it must perform almost a squared number of operations.

8.1.3 Benchmarking conclusions

One should be cautious about overemphasising the meaning of these results. The hardware architectures are in many ways incomparable, and the results are very specific and do not apply to all programming situations. Still, they may serve as a rough guide to how programming for the XScale CPU on the iPAQ family of PDAs is best carried out. A given CPU architecture will strive towards the highest possible speeds and use of faster cache helps greatly here. In the case of time critical applications with high computational requirements, a workstation will probably, if it is an option, provide a better alternative.

As high-end, newer CPUs come with clock speeds of over 3GHz, L2 caches up to and over a megabyte, and L1 caches of up to and over 128kB, the XScale can not really compete. PDAs are not designed for high performance work, mainly as the hardware needs to be compact and power saving. The decision of dividing the monitoring application into a client and server with CPU and memory intensive operations handled by the (workstation) server and the (PDA) client working as a portable user interface thus seems to be strengthened by these results.

Chapter 9

Conclusion and future work

A rapid development of technical aids is going on in the fields of patient treatment and hospital administration. As computers become more powerful, tasks earlier performed by specialised hardware can now more and more be performed in software, reducing the cost and time of development and increasing the possibilities of computer systems.

As patient monitoring equipment incorporates standardised output systems, like ethernet ports, the opportunities for computerised treatment of the data is raised to levels not possible only a few years ago. Where a given monitor formerly was limited to display its data on the built-in screen, it can now pass the data along to the hospital network where it can be input to any kind of software.

This will undoubtedly pave the way for many new ways of interpreting patient data, where for example automated diagnostic software taking into account multiple types of data in the future is likely to provide good decision support tools.

9.1 The monitoring application

With the rigorous safety requirements imposed on technological equipment by hospital regulations, the monitoring application is still far from a state in which it could be tested in practise. At the time of writing, the following can be demonstrated:

- Patient data can be transferred to and displayed on the device.

- QRS detection can be performed on a server and heart rate can be displayed on the device.
- Threshold values can be set on the device, causing an alarm to be sounded and the plot colour to change if the heart rate comes out of the specified range.
- A short history is kept on the device, permitting the user to scroll back to review the latest seconds.

The XScale provides a proper application platform, but, apparently, has some limitations, particularly with regard to floating point calculations. When programming time critical applications for this platform, this type of operations should be avoided as much as possible.

9.2 Platform and operating system

With a multitude of different devices running different operating systems, there is no obvious best choice. However, with the current (at the time of writing) state of support for the different hardware components in the iPAQ H5450, this device with Linux may not be a viable alternative for large scale implementation. There are other iPAQ models on which Linux support may be better, but, as I have had no access to these models, any concrete recommendations can not be made.

Another option is Sharp's Zaurus, which runs QTopia, the QT Embedded version OPIE is also derived from. Although this is not attempted, making the developed application work on QTopia should be very simple and only require rewriting a few lines of code. As the Zaurus PDA family comes with complete hardware support for QTopia, this alternative might be more stable than Familiar Linux on iPAQ.

The Zaurus can also be used with OPIE, under the OpenZaurus Linux distribution, if free, open source software is desirable.

At the current state of development, a rewrite would not prove very costly. The option of moving to other platforms than Linux is of course an option. Both Windows and Palm OS are viable alternatives and should be evaluated as possibilities and both these alternatives come with their own sets of possible PDAs.

In particular, the option of using LabVIEW for development should be evaluated for development of a production quality system as newer PDA versions of this system continue to increase its versatility.[1]

A continued focus on open and free systems can, however, provide a basis for a medical toolkit that is independent of proprietary software systems and can be benefit in areas where software licensing costs are prohibitive, like development countries, some educational environments and non-profit organisations.

A collection of open source software free from licencing fees could provide a pool of components, reduce the need for reimplementing the same solutions and enable developers and medical environments to focus on expanding the range of tools and their functionality at a greatly reduced cost.

This is especially true with medical applications where reliability requirements enforce a rigid testing and approval system. If a suitable reusable component was preapproved, this could help reduce implementation time of new solutions.

Focusing on free, open source systems components could also help improve standardisation, as this might make it easier for systems from different manufacturers to communicate since they could all share, at least partly, the same code and protocols.

9.3 Further work

While the direction of this project can by no means be laid out at the current time, some recommendations can be made:

- Evaluation of the system should be carried out, in conjunction with medical personnel. Due to a shortage of time, this could not be carried out in this thesis.
- The QRS detection algorithm should be evaluated and possibly be swapped for another if necessary.

[1] The Interventional Centre at Rikshospitalet has also participated in a project where an application is developed using LabVIEW where patient data is transferred to a PDA over a network connection. This system is, however, not real-time and thus serves a somewhat different purpose than the monitoring application developed for this thesis.

- Extended support for ECG signal processing should be implemented.
- Automated interpretation of other types of signals should be implemented, for example blood pressure.
- The deviation in the implementation of the QRS detection algorithm explained in Section 6.1.3 should be investigated further.

Besides these recommendations, some suggestions with regard to the possibilities presented by a roaming monitor can be mentioned:

A system of prescription approval should be evaluated. With the help of electronic signature methods, like, e.g., PGP, a physician could identify himself or herself and approve prescriptions without being in the same place as the person making the request. This could reduce the risk of embezzlement of medications (with automatic, electronic logging systems) as well as the risk of wrong medication or dosage being given, for example for some classes of medications, multiple “signatures”, or the approval of the chief physician, could be required. All this with reduced overhead for the staff, as, for all participants in the signing procedure, it could be carried out instantly, anywhere.

The current history functionality is currently constrained to the client. This means that if this history was to be allowed to grow in size, it would quickly deplete the PDAs memory. However, an opportunity to review older history on the PDA would expand the application areas for the device and such functionality should be considered. This could easily be accomplished by logging all the data on the server and transferring it to the PDA on request.

Bibliography

[1] André D. Balsa. *Linux Benchmarking HOWTO*, August 1997. http://www.tldp.org/HOWTO/Benchmarking-HOWTO.html.

[2] Ivaylo I Christov. Real time electrocardiogram qrs detection using combined adaptive threshold. *BioMedical Engineering OnLine*, August 2004.

[3] Den Norske Lægeforening. *Standard for intensivmedisin*, 2 edition, April 2001.

[4] Barbara J. Drew, Robert M. Califf, Marjorie Funk, Elizabeth S. Kaufman, Mitchell W. Krucoff, Michael M. Laks, Peter W. Macfarlane, Claire Sommargren, Steven Swiryn, and George F. Van Hare. Practice Standards for Electrocardiographic Monitoring in Hospital Settings: An American Heart Association Scientific Statement From the Councils on Cardiovascular Nursing, Clinical Cardiology, and Cardiovascular Disease in the Young: Endorsed by the International Society of Computerized Electrocardiology and the American Association of Critical-Care Nurses. *Circulation*, 110(17):2721–2746, 2004.

[5] Fiorenzo Gaita, Carla Giustetto, Francesca Bianchi, Christian Wolpert, Rainer Schimpf, Riccardo Riccardi, Stefano Grossi, Elena Richiardi, and Martin Borggrefe. Short QT Syndrome: A Familial Cause of Sudden Death. *Circulation*, 108(8):965–970, 2003.

[6] D. Geer. Survey: Embedded linux ahead of the pack. *Distributed Systems Online, IEEE*, 5(10):3–3, October 2004. Digital Object Identifier 10.1109/MDSO.2004.28.

[7] Jamey Hicks. *How to Run Linux on iPAQ Handhelds*. Compaq, February 2003.

[8] Jan Olav Høgetveit. *En datapresentasjonsform spesialtilpasset intensivmedisinen*. 1997.

[9] Michal Lauer. Building embedded linux distributions with bitbake and openembedded. Technical report, OpenEmbedded, 2005. http://www.vanille.de/tools/FOSDEM2005.pdf.

[10] A Lennon. Embedding linux. *IEE Review*, 47(3):33–37, May 2001.

[11] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition edition, 2000.

[12] Statens Strålevern. Strålevernshefte 22. http://www.nrpa.no/.

[13] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education International, fourth edition, 2003.

[14] *Linux initrd(4) man page*, November 1997.

[15] http://en.wikipedia.org/wiki/ECG.

[16] http://en.wikipedia.org/wiki/Embedded_system.

[17] http://en.wikipedia.org/wiki/Framebuffer.

[18] http://en.wikipedia.org/wiki/Heart.

[19] http://en.wikipedia.org/wiki/JFFS2.

[20] http://en.wikipedia.org/wiki/Journaling_file_system.

[21] http://handhelds.org/hypermail/ipaq/108/10806.html.

[22] http://oe.handhelds.org/.

[23] http://www.ecglibrary.com/.

[24] http://www.gtk.org/.

[25] http://www.hpl.hp.com/research/crl/index.html.

[26] http://www.palmos.com/.